Ruben Schuitemaker

Quantum money from knots

Bachelor thesis

30 June 2023

Thesis supervisors: **dr. P.J. Bruin**
dr. A.W. Laarman

Leiden University
Mathematical Institute

Abstract

Quantum computing has the potential to revolutionise the field of cryptography. Quantum money is a cryptographic scheme that attempts to create unforgeable currency. This thesis investigates the knot-based quantum money scheme proposed by Farhi et al.[FGH+12], which assumes that finding transformations between equivalent knots is computationally demanding. We start by providing a comprehensive understanding of the relevant concepts of knot theory, particularly the Alexander polynomial. Next, we discuss the proposed quantum money scheme. Finally, we discuss implementation challenges on a quantum simulator.

Contents

1 Introduction

Even though quantum computers are still in their developmental stage, they can potentially create significant advancements in various scientific fields, including cryptography, healthcare, and material sciences. One particularly captivating application is the concept of quantum money, which utilises the principles of quantum mechanics, such as the no-cloning theorem, to create secure, unforgeable currencies. This thesis examines the current work on quantum money schemes. In particular, it explores the paper 'Quantum money from knots' by Farhi et al. [FGH+12]. In 2012, the authors of this paper proposed a quantum money scheme with security based on the assumption that given two different-looking but equivalent knots, it is difficult to find an explicit transformation that takes one to the other. To do so, it leverages a particular topological invariant, the Alexander polynomial.

First, this paper introduces some basic definitions of knot theory, working towards a rigorous definition of the Alexander polynomial. Next, we briefly discuss some basics of quantum mechanics, the idea of public-key quantum money, followed by an outline of the proposed knot-based quantum money scheme. How is 'quantum money' generated and verified? We conclude by discussing the challenges of implementing such a scheme on a quantum simulator.

Quantum money has the potential to revolutionise financial transactions by offering unparalleled security compared to classical cryptographic methods. For example, traditional currency systems and cryptocurrencies still rely on classical cryptography, which may be vulnerable to attacks from increasingly powerful quantum computers. Moreover, cryptocurrencies are criticised for their high energy consumption and inefficiency, stemming from the computationally intensive proof-of-work protocols required for transaction validation. In contrast, quantum money schemes based on knot theory may offer a more energy-efficient and secure alternative. One key advantage of the proposed knot-based quantum money scheme is the potential for a polynomial time public verification algorithm, eliminating the need for a middleman in the verification process and leading to increased efficiency and reduced transaction costs.

The proposed knot-based quantum money scheme remains unbroken, making it a promising avenue for research. Understanding the mathematical foundations of quantum money can have profound implications for secure communications, financial systems, and digital currencies, whether or not the scheme is ultimately proven to be secure. Exploring the security of this scheme can help guide the

development of future quantum-resistant technologies while also investigating potential energy-efficient alternatives to existing currency systems.

Stephen Wiesner introduced the concept of quantum money in 1969, proposing using the no-cloning theorem to generate bills that can not be copied. However, Wiesner's original definition had a limitation: only the mint that produced the quantum money state could verify it. In recent years, there has been a growing interest in designing quantum money that can be verified by anyone with a quantum computer, also known as public-key quantum money. However, such money cannot be information-theoretically secure and must rely on computational assumptions.

Aaronson showed in [AC12] that public-key quantum money exists relative to a quantum oracle and proposed a concrete scheme without an oracle, but later it was broken. An oracle, in the context of theoretical computer science, is an abstract entity that can answer specific questions or solve specific problems in a way that is beyond the capabilities of the algorithm or machine requesting the information. In the case of quantum money, think of a quantum oracle as a 'black box' that aids in the design or security of the quantum money scheme. The main open question in the field is how to design secure quantum money that does not require an oracle and does not need the mint's participation to use the money. It was proposed in [LAF+09] to use quantum money that is not based on a classical secret but instead on the hardness of generating a known superposition. However, they did not present a complete proposal for a quantum money scheme, only a blueprint.

The primary goal of this thesis is to provide a comprehensive understanding of the algebraic definition and properties of the Alexander polynomial as a link invariant and its application in the context of quantum money from knots.

This thesis is organised as follows:

1. Definition and properties of links, working towards the definition of the Alexander polynomial.
2. A short introduction to some basic concepts from quantum mechanics
3. Exploration of the quantum money from knots scheme: how can we generate and verify 'quantum money'?
4. Discussion of a simplified implementation of this scheme using quantum simulation software.

2 Topological preliminaries

2.1 Knots and links

In this section, we introduce the concept of a mathematical knot, drawing inspiration from [Lic12], [BZH13], [Rol03]. A basic understanding of topology is assumed, particularly the theory of fundamental groups and covering spaces, as outlined in [Bru21].

In topology, knot theory studies mathematical knots inspired by natural everyday knots. Unlike a shoelace, a mathematical knot has its ends joined to form a closed loop in 3-dimensional space. The simplest knot is called the unknot, which is just a ring. Our knots also have an orientation: a preferred direction around the closed loop.

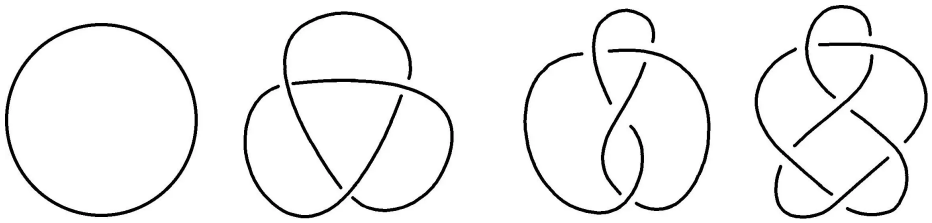


Figure 1: The unknot, the trefoil, the figure eight knot and the 6_2 knot

As is customary, $\mathbb{R}^n$ will denote n-dimensional Euclidean space, and S^n will be the n-sphere. We can thus think of S^n as the unit sphere in $\mathbb{R}^{n+1}$.

Definition 2.1.1. A *link* of n *2* $\mathbb{Z}_{>0}$ components is a continuous embedding $\bigsqcup_{i=1}^{n} S^1 \hookrightarrow \mathbb{R}^3$ of the disjoint union of n circles into n disjoint closed curves in $\mathbb{R}^3$. Each component of the link is equipped with an orientation induced by considering the standard orientation of S^1 (counter-clockwise). The components are ordered. A 1-component link is called a *knot*.

Taking the embedding to S^3 or some other 3-manifold is also possible. One motivation is the fact that S^3 is compact. However, we will not use this, so, for simplicity, we will be using $\mathbb{R}^3$.

Topologically, any two links with the same number of components will always be homeomorphic. In particular, any two knots are homeomorphic. An injection $S^1 \hookrightarrow \mathbb{R}^3$ induces a bijection to its image. Since this is a continuous map from a compact space to a Hausdorff space, it is a homeomorphism.

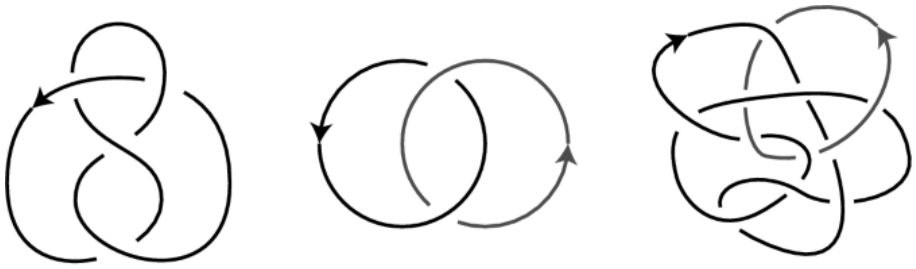


Figure 2: Examples of links

We, of course, want a way to distinguish different links. Therefore, a different notion of equivalence between links is required. One that does not allow the knot to cross over itself while deforming.

We will consider two links equivalent if they have the same number of components and can be transformed into each other through a continuous deformation of the ambient space without cutting or passing through the link. We also do not want to distinguish between links that differ by reordering the components.

Definition 2.1.2. Let L and $L^{\prime}$ be two links of n components. We say that L and $L^{\prime}$ are equivalent or *ambient isotopic* if there exists a continuous map

$$F\colon [0,1] \quad \mathsf{R}^3 \;\; ! \;\; \mathsf{R}^3,$$

with the following properties

- $F(0, \quad)$ is the identity on R^3.
- $F(t, \quad)$ a homeomorphism for every $0 \quad t \quad 1$.
- $F(1, \quad)$ preserves the orientation of R^3 and $F(1, \quad) \quad L \quad \sigma = L^{\prime}$. Here, $\sigma\colon \bigsqcup_{i=1}^{n} S_i^1 \;\; ! \;\; \bigsqcup_{i=1}^{n} S_{\tau(i)}^1$ is a reordering of circles induced by a permutation $\tau \; 2 \; S_n$.

The map F is called an *ambient isotopy* taking L to $L^{\prime}$. It preserves the orientation but not the ordering of its components.

This definition gives us an equivalence relation on links: two links are equivalent if and only if an ambient isotopy exists between them.

Remark 2.1.1. A choice can be made whether we want to consider links with reordered components as equivalent. If we want to distinguish different orderings of the components, we will have link diagrams that look entirely identical but represent equivalent links that are not equivalent.

Example 2.1.1. Consider two different links of 2 components consisting of an unknot linked with a trefoil knot that only differs in their ordering of the components. Do we want these to be considered equivalent?

Another definition we need concerns how well-behaved a link is.

Definition 2.1.3. We call a link *polygonal* if each of its components can be represented by a finite closed polygonal chain: a finite set of straight line segments connected to form a closed loop. We call a link *tame* if it is equivalent to a polygonal link; otherwise, it is called *wild*.

Figure 3 shows an example of a wild knot. Intuitively it is clear that this knot with an infinite amount of crossings can not be represented by a polygonal knot.

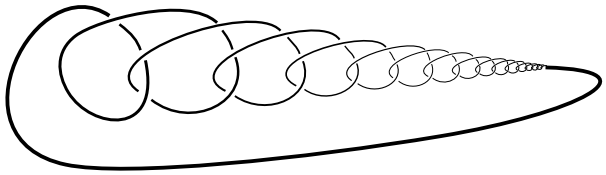


Figure 3: Example of a wild knot

As opposed to tame links, wild links often display pathological behaviour. They provide counterexamples to general versions of theorems which hold for the tame knots. Certain invariants, namely, are not necessarily defined for a wild link. In this text, we want to avoid this wildness, so from now on, all links here are assumed to be tame.

Definition 2.1.4. Let L be a link of n components. A *projection* of L is a pair (π, O) consisting of

- A projection $\pi\colon \mathrm{im}(L) \to \mathbb{R}^2$ induced by the standard projection $\mathbb{R}^3 \to \mathbb{R}^2,\ (x,y,z) \mapsto (x,y)$.
- An *orientation assignment* O: a choice of one of two possible orientations (positive or negative) for each of the n closed curves in $\mathrm{im}(L)$. A curve is positively oriented if a counterclockwise walk of S^1 induces a counterclockwise walk of the closed curve under the projection π.

Such a projection of L is called *regular* if

- The set of crossings $X = \{x \in \mathbb{R}^2 : |\pi^{-1}(x)| > 1\}$ is finite.
- For every $p \in \mathrm{im}(L)$ we have $|\pi^{-1}(x)| \leq 2$ with equality precisily if $p \in X$.

- For each crossing x, the curves crossing at x are not tangent but cross over each other.

Our choice to avoid wild links gives us the following pleasant result.

Proposition 2.1.1. Every equivalence class of links contains a representative L which admits a regular projection.

Proof. See [Liv93] for knots and [Sto] for a full proof for links. □

Note that information is lost by taking a regular projection of a link. In particular, the relative height of the two points in the fibre $\pi^{-1}(x)$ of a crossing x is forgotten. Remembering this distinction as well leads us to the following definition.

Definition 2.1.5. Let L be a link of n components. Then, a *link diagram* of L is a triple (π, O, C) consisting of a regular projection (π, O) of L together with an *crossing assignment* of the crossings of $\mathrm{im}(\pi)$: a choice of which curve lies above the other. This data can be determined by L.

Determining C from L is relatively easy. Let n be the natural normal vector to the plane. Then if $x, y \in \mathrm{im}(L)$ are distinct points such that $\pi(p) = \pi(q)$, we must have $p - q = cn$ for some non-zero $c \in \mathbb{R}$. If $c > 0$, then p and its curve segment lie above q and its curve segment. If $c < 0$, we, of course, have the opposite.

This definition is helpful for visualising links; we have already seen some of these diagrams earlier.

Remark 2.1.2. Every link has a link diagram. Conversely, a link diagram also determines a link up to ambient isotopy. As long as we know the crossing assignments, the distance between the lines of the crossing in the link can be adjusted via ambient isotopy.

We have a notion of equivalence for link diagrams analogous to the notion of ambient isotopy of links discussed earlier.

Definition 2.1.6. Let (π, O, C) and (π', O', C') be two link diagrams. We call them equivalent or *planar isotopic* if there exists a continuous map

$$F\colon [0,1] \times \mathbb{R}^2 \to \mathbb{R}^2,$$

with the following properties

- $F(0, -)$ is the identity on $\mathbb{R}^2$.
- $F(t, -)$ a homeomorphism for every $0 \leq t \leq 1$.

- $F(1, \quad) \ \pi = \pi^{\emptyset}$, such that the orientation and crossing assignments induced by $F(1, \quad)$, O and C agree with $\mathsf{O}^{\emptyset}$ and $\mathsf{C}^{\emptyset}$.

The map F is called an *planar isotopy*.

The above gives us an equivalence relation on link diagrams: two diagrams are equivalent if and only if a planar isotopy exists between them. However, envisioning and thinking about link diagrams is much simpler than links. Therefore, it would be very beneficial to have the necessary and sufficient conditions for determining the equivalence of links based on their corresponding diagrams. Fortunately, such a condition does indeed exist.

Definition 2.1.7. Two link diagrams are equivalent if and only if a finite sequence of Reidemeister moves exists, taking one to the other. There are 3 different moves (6 including inverses). Figure 4 displays the three types of moves (and their inverses).

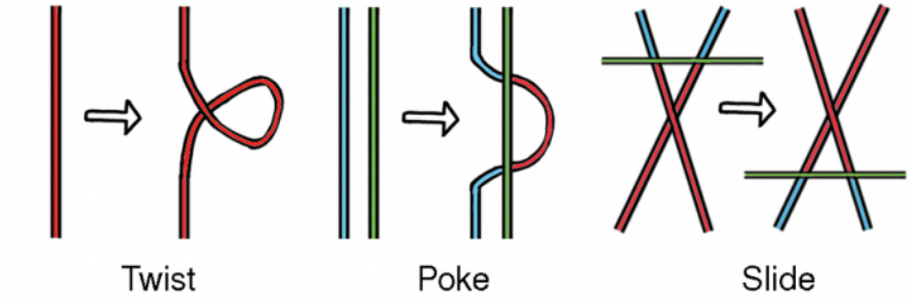


Figure 4: The three types of Reidemeister moves

One 'small' drawback is that the best lower bound on the number of Reidemeister moves required is rather 'large' [CL14].

We can now characterise link equivalence in terms of their diagrams!

Theorem 2.1.1. (Reidemeister's Theorem) Let L and $L^{\emptyset}$ be links with link diagrams $(\pi, \mathsf{O}, \mathsf{C})$ and $(\pi^{\emptyset}, \mathsf{O}^{\emptyset}, \mathsf{C}^{\emptyset})$ respectively. Let D and $D^{\emptyset}$ denote the equivalence classes of these diagrams modulo planar isotopy. Then L and $L^{\emptyset}$ are equivalent (ambient isotopic) if and only if D and $D^{\emptyset}$ are equivalent.

Proof. The main idea is that of subdivision. Working piecewise-linearly, one can reduce to the case of only considering so-called minimal moves on link diagrams. A finite sequence of Reidemeister moves can describe these minimal moves. See for example section 2.1 [Kau05] or 4.1.1 from [MK96]. □

The following definition gives another way to specify a diagram of a link, one that is more suitable to implement on a computer.

Definition 2.1.8. A link can also be represented by a so-called *planar grid diagram*: a $d \quad d$ grid on which we place d X's and d O's. There must be precisely one X and one O in each row and each column, and there may never be an X and an O in the same cell.

Horizontal arrows from O to X are drawn in each row. Vertical arrows are drawn from X to O. Where horizontal lines and vertical lines intersect, the vertical line always crosses above the horizontal line.

A planar grid diagram G can be specified by two disjoint permutations $\pi_X, \pi_O \in S_d$, in which case the X's have coordinates $\{(i, \pi_X(i))\}$ and the O's have coordinates $(i, \pi_O(i))$ for $i \in \{1, \ldots, d\}$. Two permutations are said to be disjoint if, for all i, we have $\pi_X(i) \neq \pi_O(i)$. Any two disjoint permutations $\pi_X, \pi_O \in S_d$ thus define a planar grid diagram $G = (\pi_X, \pi_O)$. Every link can be represented by many different grid diagrams.

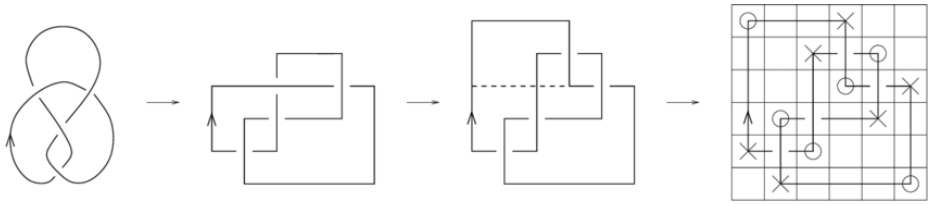


Figure 5: Transformation of a projection of the figure eight knot to a planar grid diagram

Implementation-wise, it will be convenient to use planar grid diagrams and store links as two disjoint permutations, which can be represented as a single bitstring. We will also need a way to apply Reidemeister-type moves to these diagrams. Analogous to the Reidemeister moves, we have the so-called Cromwell moves. These three types of moves (transformations) on planar grid diagrams are sufficient to generate all planar grid diagrams of the same link equivalence class.

The three types of Cromwell moves are translation, commutation, and stabilisation/destabilisation. We shall define each type of move as follows.

Translation is either vertical or horizontal. A vertical translation moves the top row of the diagram to the bottom of the diagram or vice versa while leaving the rest unchanged. Horizontal translation, similarly, moves the leftmost column of the diagram to the rightmost or vice versa. Note that the class of grid diagrams

modulo this translation action can be naturally identified with grid diagrams on a torus.

Commutation interchanges two adjacent rows or two adjacent columns. We can apply commutation in two cases: the line segment defined by the X and O of one row (or column) must be strictly contained in or disjoint from the line segment of the other row (or column).

The last type of Cromwell move is slightly more involved. An X (resp. O) destabilisation replaces a 2 2 subgrid containing two X's and one O (resp. two O's and one X) with a single square containing an X (resp. O), removing one row and one column in the process. Stabilisation is the inverse of destabilisation. Each (de)stabilisation is identified by its type, X or O, along with the corner in the 2 2 subgrid not occupied by a symbol. We get eight possibilities: X:NW, X:NE, X:SW, X:SE, O:NW, O:NE, O:SW, O:SE.

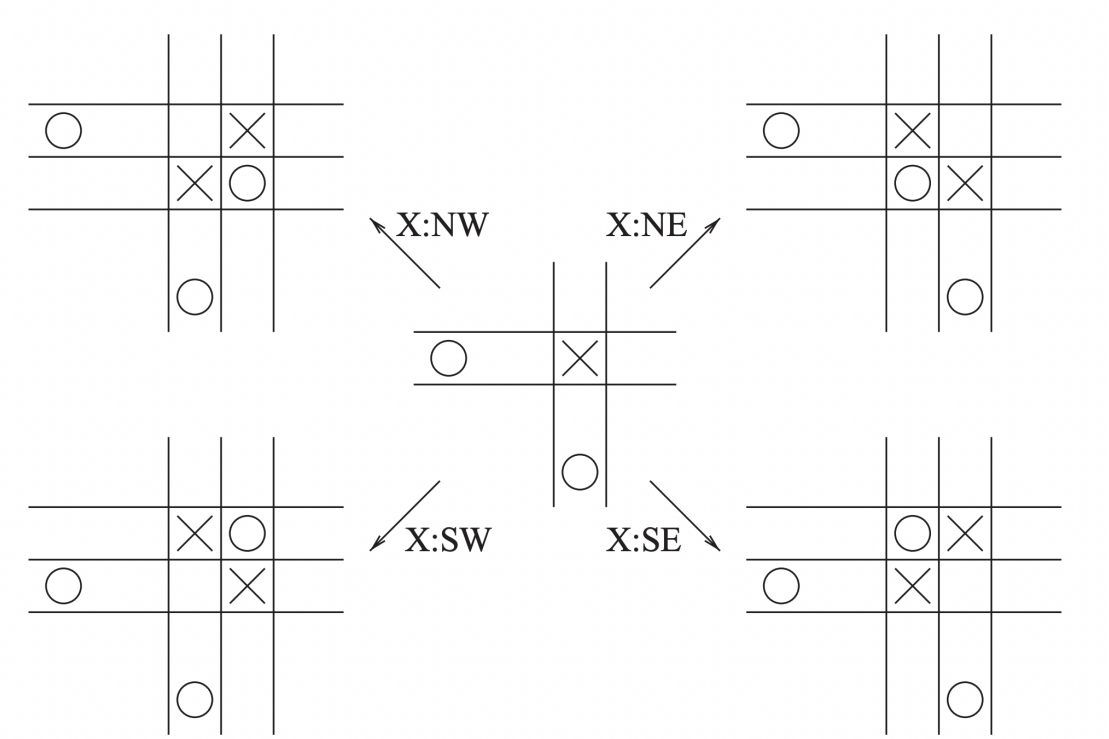


Figure 6: The four types of X (de)stabilisations grid moves

Theorem 2.1.2. Two planar grid diagrams represent the same link equivalence class if and only if there exists a finite sequence of translation, commutation, and (de-)stabilisation grid moves to relate one grid to the other.

Proof. See [Cro95] and also [Dyn06]. □

2.2 The link group

Our primary goal in this chapter is to define an invariant of links. A link invariant is a map from equivalence classes of links to some mathematical structure. The key requirement here is that it assigns equivalent links to the same value. There are many different link invariants one could define, but we will focus on one in specific, the Alexander polynomial. One reason for this choice is that this invariant is well-understood and relatively easy to compute. Therefore, it is very suitable for our application. To start defining the Alexander polynomial, we will first take a look at a more basic invariant of a link: the link group.

Definition 2.2.1. A *tubular neighbourhood* V_L of L is a neighbourhood of L homeomorphic to a disjoint union of solid tori, one for each link component containing that particular component in its interior.

Definition 2.2.2. Let L be a link and consider one of its components, K. A simple closed curve m exists on the boundary ∂V_K, which is null-homotopic in V_K but not on ∂V_K. We call m a *meridian* of K. It is equipped with a standard orientation induced by L.

Note that any two meridians of a knot K are homotopic in ∂V_K.

Definition 2.2.3. Let L be a link. We call $X_L := \mathbb{R}^3 \setminus \mathrm{im}(L)$ the *link complement* in $\mathbb{R}^3$.

Definition 2.2.4. Let L be a link. We define the *link group* G_L as the fundamental group of the link complement, $G_L := \pi_1(X_L)$.

Proposition 2.2.1. The link group is well-defined and is an invariant of the link.

Proof. Note that we do not require a base point since the complement of a (tame) link is path-connected. For example, suppose L_1 and L_2 are equivalent links. Then, by definition, there exist a homeomorphism $\mathbb{R}^3 \to \mathbb{R}^3$ taking $\mathrm{im}(L_1)$ to $\mathrm{im}(L_2)$, so their complements must be homeomorphic too. □

2.3 The Wirtinger presentation

Given a link diagram of some link L, there is a straightforward method to obtain a presentation for the link group. Divide the diagram into i segments with breaks occurring at under-passes. For each such segment, we take a group generator g_i. Corresponding to each crossing c of the diagram, we add a relation r_c. Suppose at the crossing c the over-pass arc is labelled g_k and the under-pass is labelled g_i; as it approaches c and g_j as it leaves c. Then we take $r_c = g_k g_i g_k{}^{1} g_j{}^{1}$ if the sign of the crossing is negative, that is, g_k has orientation towards the left as we approach along g_i, see Figure 7. If the sign is positive, we take $r_c = g_k{}^{1} g_i g_k g_j{}^{1}$ instead. The resulting presentation is called the Wirtinger presentation of the link group.

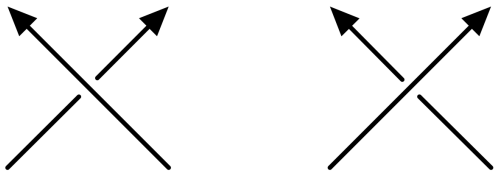


Figure 7: Crossings with negative and positive sign

Note that equating a relation r_c to the identity is the same as saying that g_i and g_j are conjugate by means of g_k or $g_k{}^{1}$.

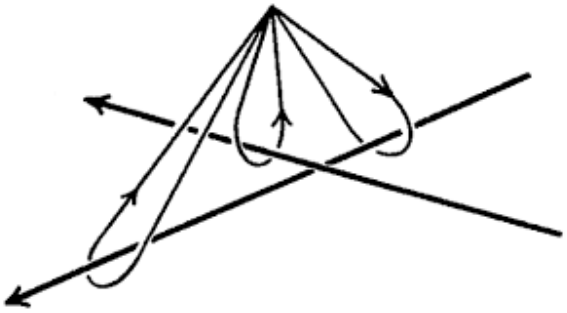


Figure 8: Representations of generators of a link group.

One can image each generator g as a loop starting from a base point above the diagram, encircling the i^{th} segment in a positive direction and returning

immediately to the base point, as shown in Figure 8. In this context, the relations we add ensure that the concatenation of g_k, g_i and g_k^{-1} is path homotopic to g_j.

Theorem 2.3.1. The link group is generated by the (homotopy classes of the) loops g_i, and it has presentation

$$G_L = \langle g_1, \dots g_m | r_1, \dots r_n \rangle,$$

the *Wirtinger presentation*. We have $m = n$ unless L contains a component with no underpasses.

Proof. Intuitively this is clear as shown in Figure 8. A proof can be found in [BZH13, Theorem 3.4]. □

Corollary 2.3.1. Let L be a link and $\langle g_1, \dots g_m | r_1, \dots r_n \rangle$ a Wirtinger presentation of G_L. Then each relation r_i is a consequence of the other relations. It follows that a Wirtinger presentation with m generators requires, at most, $m \quad 1$ relations.

Proof. See [BZH13, 3.6]. □

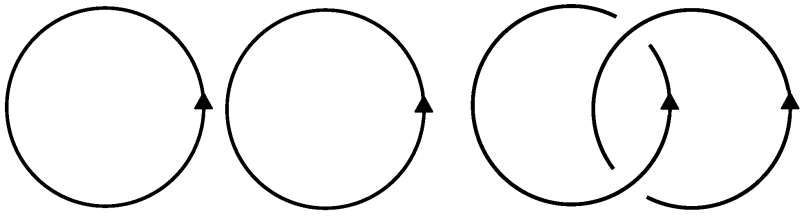


Figure 9: A trivial link and the Hopf link

Example 2.3.1. The trivial link of 2 components in Figure 9 has 2 segments and 0 crossings giving 2 generators and 0 relations between them. The Wirtinger presentation $\langle g, h | \rangle$, therefore, presents $\mathbb{Z} \quad \mathbb{Z}$, the free group of rank 2.

Example 2.3.2. We will now use this recipe to find a presentation of the Hopf link L, the simplest non-trivial link (as we will now prove). Both components have a counterclockwise orientation. The diagram consists of two segments; we label them g and h. There are 2 crossings, the first gives the relation $r_1 = g^{-1}hgh^{-1}$

and the second gives $r_2 = h\ ^1ghg\ \ 1$. These both imply that $gh = hg$. The associated group, G_L, therefore has Wirtinger presentation

$$G_L = \hbar g, h \jmath g h = h g \imath,$$

which presents $\mathsf{Z}\quad\mathsf{Z}$, the free abelian group of rank 2.

We have now shown the existence of a non-trivial link. This is because the link groups of the trivial link and the Hopf are not isomorphic, and since the link group is an invariant of links, this must mean that these links are not equivalent.

The generators of a Wirtinger presentation corresponding to a single link component belong to the same conjugacy class in the group. Further, if the group is abelianised, we would require that these conjugated elements commute. Then, the group becomes just the direct sum of copies of Z, one for each link component, with all the generators corresponding to a single link component becoming the generator of one of the copies of Z. As we will soon see, this is the first homology group of X_L. The loops representing generators of G_L in the Wirtinger presentation also represent meridian generators of $H_1(X_L)$.

2.4 The Alexander polynomial

It is now almost time to define the Alexander polynomial. However, first, we will need a couple more definitions.

In order to better understand the link group, we can explore its structure by looking at a specific covering of the link complement, the so-called infinite cyclic cover. For this, we will need some information about the link group. Here is a key fact that we will use.

Lemma 2.4.1. Let L be a link of n components. Then we have $H_1(X_L) = G_L^{\mathsf{ab}} = \bigoplus_{i=1}^{n} \mathsf{Z}$. In other words, the abelianisation of the link group is isomorphic to the direct sum of n copies of Z.

Proof. This proof will make use of some homology theory. Consider a tubular neighbourhood V of L in R^3. Together with the link complement X_L, we have $X_L \ [\ V = \mathsf{R}^3$, so we can apply the theorem of Mayer-Vietoris to obtain the following exact sequence of homology groups.

$$\ldots \quad ! \ H_2(\mathsf{R}^3) \quad ! \ H_1(X_L \setminus V) \quad ! \ H_1(X_L) \quad H_1(V) \quad ! \ H_1(\mathsf{R}^3) \quad ! \ \ldots$$

The first and second homology groups of R^3 are trivial. The intersection $X_L \setminus V$ of the complement and a tubular neighbourhood of a link is homotopy equivalent

to the union of n (non-solid) tori. This gives us that $H_1(X_L \setminus V) = \mathbb{Z}^{2n}$. The tubular neighbourhood V is homotopy equivalent to the disjoint union of n circles, this yields $H_1(V) = H_1(\bigsqcup_{i=1}^{n} S^1) = \mathbb{Z}^n$. We obtain the exact sequence

$$0 \quad ! \quad \mathbb{Z}^{2n} \quad ! \quad H_1(X_L) \quad \mathbb{Z}^n \quad ! \quad 0,$$

so we see that $\mathbb{Z}^{2n} = H_1(X_L) \quad \mathbb{Z}^n$ and $H_1(X_L) = \mathbb{Z}^n$. Finally, by the theorem of Hurewicz, we obtain

$$H_1(X_L) = \pi_1(X_L)^{\mathsf{ab}} = G_L^{\mathsf{ab}},$$

as desired. □

Proposition 2.4.1. The homology classes of the meridians of the individual components of a link generate $H_1(X_L) = \bigoplus_{i=1}^{n} \mathbb{Z}$.

Proof. The first homology group $H_1(X_L)$ can be obtained by abelianising a Wirtinger presentation for G_L. From the construction described earlier, it seems reasonable to assume that the natural map $G_L \; ! \; G_L^{\mathsf{Ab}} = H_1(X_L)$ maps each generator to a meridian of the corresponding component. This is discussed in depth in [Lic12, Theorems 1.5 and 1.7]. □

We are now ready to define interesting coverings of link complements. Let L be a link of n components and consider a surjective group homomorphism $\phi\colon G_L \; ! \; F$ to a free abelian group $F = \bigoplus_{i=1}^{m} \mathbb{Z}$. The link complement X_L is semi-locally simply connected and thus admits a universal covering [RSdJ22]. The Galois correspondence of covering spaces shows the existence of a connected regular cover $p\colon Y \; ! \; X_L$ such that $p\ (\pi_1(Y)) = \ker \phi$. We then have

$$\mathrm{Aut}(Y/X_L) = \pi_1(X_L)/p\ (\pi_1(Y)) = G_L/\ker \phi = F.$$

Observation 2.4.1. As Y is connected, the induced homomorphism p is injective; this means we can consider the fundamental group of Y as a subgroup of $\pi_1(X_L)$ and therefore $p\ (\pi_1(Y)) = \pi_1(Y)$ holds. By Hurewicz, we have $H_1(Y) = \pi_1(Y)^{\mathsf{ab}} = (\ker \phi)^{\mathsf{ab}}$. We see that $H_1(Y)$ can be understood as a quotient of a normal subgroup of G_L.

Definition 2.4.1. Let L be a link of n components and consider the composition of homomorphisms

$$G_L \quad ! \quad G_L^{\mathsf{ab}} = \bigoplus_{i=1}^{n} \mathbb{Z} \quad ! \quad \mathbb{Z},$$

defined by the canonical quotient map followed by mapping each meridian to 1. Call this surjective composition ϕ. We have a covering map

$$p\colon X_1 \to X$$

corresponding to $\ker\phi$ with automorphism group

$$\mathrm{Aut}(X_1/X_L) = \pi_1(X_L)/p_*(\pi_1(X_1)) = G_L/N = \mathbb{Z}.$$

We call X_1 the *infinite cyclic cover* of X_L.

There is an action of $\mathrm{Aut}(X_1/X_L) = \mathbb{Z} = \langle t\rangle$ on X_1. Any element t of this automorphism group induces an automorphism t_* on homology; we, therefore, obtain an action of $\langle t\rangle$ on $H_1(X_1)$ as well. The ring $\mathbb{Z}$ acts on any abelian group, so the group-ring $\mathbb{Z}[\langle t\rangle]$ acts on $H_1(X_1)$. Recall that for any group G written multiplicatively, the group-ring $\mathbb{Z}[G]$ refers to a collection of $\mathbb{Z}$-linear combinations of elements of G. The addition operation in $\mathbb{Z}G$ is defined by formal addition, while multiplication is determined by the distributive law and the multiplication operation in G. In this context, the ring $Z\langle t\rangle$ can be viewed as the ring $\mathbb{Z}[t^{-1}, t]$ of Laurent polynomials in t. Using this action, we can turn $H_1(X_1)$ into a module over the ring $\mathbb{Z}[t^{-1}, t]$.

Definition 2.4.2. Let L be a link. The Alexander module A of L is the $\mathbb{Z}[t^{-1}, t]$-module $H_1(X_1)$.

As we will see, this definition of the Alexander module gives rise to the single variable (reduced) Alexander polynomial. One could also define a multiple-variable Alexander polynomial, but this requires looking at different coverings of the link complement, resulting in a slightly different definition of the Alexander module.

Definition 2.4.3. Let R be a commutative ring. A *presentation* for an R module M is a short exact sequence

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

of R-modules where F and E are free. The map α will also be called a presentation. We say that M is *finitely presented* if F and E can be taken to be finitely generated. A *presentation matrix* for a finitely presented module M is a matrix representing α with respect to fixed bases of F and E.

Definition 2.4.4. Let M be a finitely presented module over a commutative ring R. Suppose A is an $m \times n$ presentation matrix of M. The r^{th} elementary ideal (or Fitting ideal) E_r of M is the ideal of R generated by all the $(m-r+1)\times(m-r+1)$ minors of A.

Proposition 2.4.2. The r^{th} elementary ideals are all invariant under a change of presentation of A.

Proof. See [CF12] □

Now we have all the tools to define the Alexander polynomial.

Definition 2.4.5. An *Alexander matrix* of a link L is a presentation matrix of its Alexander module.

Definition 2.4.6. The r^{th} *Alexander ideal* E_r is the r^{th} elementary ideal of the Alexander module M. By definition, it is generated by the $m \quad m$ minors of the $m \quad n$ Alexander matrix.

Definition 2.4.7. The r^{th} *Alexander polynomial* Δ_r *2* $\mathsf{Z}[t \ ^1, t]$ is defined to be the greatest common divisor of the elements of the r^{th} Alexander ideal. We call the first Alexander polynomial *the* Alexander polynomial written as $\Delta_L(t)$.

As we will see in the next section, the Alexander ideal exists for any knot since the Wirtinger presentation will give us an Alexander matrix of shape $m \quad n$ with $m = n \quad 1$. It is known that the elementary ideals are independent of the choice of presentation. Also, since $\Delta_L(t)$ comes from a unique factorisation ring, the gcd is well defined and determined up to multiplication by elements of $(\mathsf{Z}[t \ ^1, t])$, which are precisely the monomials $\mathsf{f} \quad t^n j n$ *2* $\mathsf{Z}g$.

2.5 Fox derivatives

The Alexander polynomial, as defined currently, is not a very tangible construct. Therefore, this subsection will describe a method for obtaining the Alexander polynomial explicitly from a presentation of the link group. More specifically, we will use the Wirtinger presentation to obtain an Alexander matrix: a presentation matrix for the Alexander module.

Definition 2.5.1. Let F be a free group on generators $\mathsf{f}g_1, g_2, \ldots, g_n\mathsf{g}$. The *Fox derivative* with respect to g_i is defined to be the map

$$\frac{\partial}{\partial g_i} : \mathsf{Z}[F] \ ! \ \mathsf{Z}[F],$$

such that for any u, v *2* F and x, y *2* $\mathsf{Z}[F]$, it obeys the following axioms

- $\frac{\partial}{\partial g_i}(g_j) = \delta_{ij}$.

- $\frac{\partial}{\partial g_i}(x+y) = \frac{\partial}{\partial g_i}(x) + \frac{\partial}{\partial g_i}(y)$.
- $\frac{\partial}{\partial g_i}(uv) = \frac{\partial}{\partial g_i}(u) + u\frac{\partial}{\partial g_i}(v)$.

In [CF12] it is proven that these axioms uniquely determine $\frac{\partial}{\partial g_i}$.

Proposition 2.5.1. These axioms imply that

- $\frac{\partial}{\partial g_i}(e) = 0$.
- $\frac{\partial}{\partial g_j}(\quad u) = \quad \frac{\partial}{\partial g_j}(u)$.
- $\frac{\partial}{\partial g_j}(u\ \ ^{1}) = \quad u\ \ ^{1}\frac{\partial}{\partial g_j}(u)$.

For an extensive treatment of the Fox derivative, see [CF12].

Definition 2.5.2. Let L be a link and let $G_L = \langle S | R \rangle$ be its finitely presented group with generators $S = \{g_1, \ldots, g_n\}$ and relations $R = \{r_1, \ldots, r_m\}$. Define the free group F on generators S. Recall from definition 2.2.1 the homomorphism

$$\phi \colon G_L \ ! \ \mathbb{Z} = \langle t \rangle,$$

sending each meridian to t. Combining this with the natural map $F \ ! \ G_L$ yields the map $F \ ! \ \langle t \rangle$. his in turn induces the map

$$\alpha \colon \mathbb{Z}[F] \ ! \ \mathbb{Z}[t\ \ ^{1}, t].$$

The *Jacobian matrix* of G with respect to the presentation $\langle S | R \rangle$ is the $m \quad n$ matrix A defined by

$$A := \left(\alpha\left(\frac{\partial r_i}{\partial g_j}\right) \right)_{\substack{1 \ \ i \ \ m \\ 1 \ \ j \ \ n}}.$$

It is dependent on the choice of presentation of G_L.

Theorem 2.5.1. This Jacobian matrix of a Wirtinger presentation of a link group X_L is an Alexander matrix. The gcd of the determinants of the $(n \quad 1) \quad (n \quad 1)$ submatrices is the Alexander polynomial.

Proof. The Alexander ideal generated by these determinants does not depend on the choice of the presentation of G_L. See [BZH13, Section 9B] for details regarding knots and 9.18 for a clarification for links. □

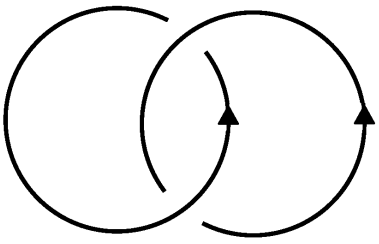


Figure 10: The Hopf link.

Example 2.5.1. We will now use these techniques to calculate the Alexander polynomial of the Hopf link L, see figure 10. As we have seen earlier, the associated group G_L has Wirtinger presentation

$$G_L = \hbar g, h\jmath gh = hgi,$$

which is the free abelian group of rank 2. In this presentation, g and h correspond to meridians of K so we have $\alpha(g) = t = \alpha(h)$. We calculate

$$\frac{\partial ghg^{-1}h^{-1}}{\partial g} = \frac{\partial gh}{\partial g} + gh\frac{\partial g^{-1}h^{-1}}{\partial g} = \frac{\partial g}{\partial g} + g\frac{\partial h}{\partial g} + gh\frac{\partial g^{-1}}{\partial g} + ghg^{-1}\frac{\partial h^{-1}}{\partial g} = 1 - ghg^{-1},$$

and

$$\frac{\partial ghg^{-1}h^{-1}}{\partial h} = \frac{\partial g}{\partial h} + g\frac{\partial h}{\partial h} + gh\frac{\partial g^{-1}}{\partial h} + ghg^{-1}\frac{\partial h^{-1}}{\partial h} = g - ghg^{-1}h^{-1},$$

so the Jacobian matrix equals

$$A = (\alpha(1 - ghg^{-1}), \alpha(g - ghg^{-1}h^{-1})) = (1 - t, t - 1,).$$

This means the Alexander ideal is generated by $(1 - t, t - 1,) = (t - 1)$ so the Alexander polynomial is $\Delta_L(t) = t - 1$.

3 Quantum preliminaries

We briefly discuss some basic concepts from quantum mechanics.

3.1 Quantum states

In quantum mechanics, quantum states are vectors in a d-dimensional complex Hilbert space H. Elements in H are called 'ket' vectors written $j\psi i \ 2 \ H$. Here ψ is just a name for the vector. These vectors can be thought of as column vectors. For example, a state $j\psi i$ in a 2-dimensional vector space with basis vectors $j0i$ and $j1i$ can be written as $j\psi i = \alpha j0i + \beta j1i$. More precisely, if H has basis $jv_1i, jv_2i, \ldots, jv_di$, we can write any quantum state in H as

$$j\psi i = \alpha_1 jv_1 i + \alpha_2 jv_2 i + \quad + \alpha_d jv_d i,$$

such that for all i, $\alpha_i \ 2 \ C$ is the amplitude corresponding to state v_i, and these amplitudes must satisfy $\sum_{i=1}^{d} j\alpha_i j^2 = 1$.

Recall that the transpose takes elements in H to its dual H by taking the inner product. The transpose of a ket vector is called a 'bra' vector and is written as $h\psi j$. These can be thought of as row vectors. We write $h\phi j\psi i$ and $j\phi ih\psi j$ for the inner and outer products. We can also combine states from two different n-qubit systems to form a joined state on $2n$ qubits by taking the tensor product. For $j\phi_A i \ 2 \ H_A$ and $j\phi_B i \ 2 \ H_B$ we often simply write $j\phi_A ij\phi_B i$ for $j\phi_A i \quad j\phi_B i$.

3.2 Measurement of quantum states

Quantum states are unable to persist indefinitely within an abstract domain; at some point, they must undergo measurement. The most elementary measurement can be carried out with respect to an orthonormal basis, represented as $jv_1i, jv_2i, \ldots, jv_di$. When measuring a quantum state $j\psi i$, the likelihood of obtaining outcome i is given by $jhv_ij\psi ij^2$. It is important to emphasise that measurement is a destructive process.

Once $j\psi i$ is measured and outcome i is observed, the state collapses to jv_ii, leading to the loss (or existence in parallel universes, depending on one's interpretation of quantum mechanics) of all other information pertaining to the original state.

It is vital to understand that measurements are the only way of accessing information about a quantum state's amplitudes. Numerous misconceptions about quantum mechanics arise from the assumption that there exists an alternative method for obtaining information about a quantum state besides measurements.

3.3 Transformations on quantum states

Measurement is a method for altering a quantum state; however, it is possible to change a state by applying a quantum transformation. All quantum transformations are linear transformations acting on quantum states. We call a linear map $U : \mathcal{H} \to \mathcal{H}$ unitary if its conjugate transpose is its inverse: $U^\dagger U = UU^\dagger = I$. Alternately but equivalently, a unitary transformation is simply a linear map that preserves the inner product between pairs of elements in $\mathcal{H}$. This, in turn, implies that a unitary transformation preserves the norm of a quantum state. Therefore, a unitary transformation maps a quantum state into another quantum state. In conclusion, all quantum operations are unitary linear transformations.

3.4 No-cloning theorem

We present a somewhat simplified version of the no-cloning theorem of quantum mechanics.

Theorem 3.4.1. An arbitrary quantum state cannot be copied. More formally, a unitary operation U on $2n$ qubits and an n-qubit state $|e\rangle$ such that for any n-qubit state $|\psi\rangle$,

$$U(|\psi\rangle|e\rangle) = |\psi\rangle|\psi\rangle,$$

does not exist.

Proof. We will try to derive a contradiction by assuming that such a unitary operator exists. Let $|\psi\rangle$ and $|\phi\rangle$ be quantum states on n qubits. Since U is supposed to clone states, we have $U|\psi\rangle|e\rangle = |\psi\rangle|\psi\rangle$ and $U|\phi\rangle|e\rangle = |\phi\rangle|\phi\rangle$. Taking the conjugate transpose gives $(U|\phi\rangle|e\rangle)^\dagger = \langle\phi|\langle e|U^\dagger$. Take the inner product and using $U^\dagger U = I$, we get

$$\langle\phi|\langle e|\psi\rangle|\psi\rangle|e\rangle = \langle\phi|\langle e|U^\dagger U|\psi\rangle|\psi\rangle|e\rangle = \langle\phi|\langle\phi|\psi\rangle|\psi\rangle = \langle\phi|\psi\rangle^2.$$

Using properties of the tensor product, along with the fact that quantum states are normalised, we also get

$$\langle\phi|\langle e|\psi\rangle|\psi\rangle|e\rangle = \langle\phi|\psi\rangle\langle e|e\rangle = \langle\phi|\psi\rangle.$$

So this means that $\langle\phi|\psi\rangle = \langle\phi|\psi\rangle^2$, meaning that $\langle\phi|\psi\rangle$ is either 0 or 1. This in turn implies that $|\phi\rangle$ and $|\psi\rangle$ are equal or orthogonal, but we have not assumed this.

Hence, there is a contradiction. Therefore, a unitary operation that perfectly clones an arbitrary state does not exist, which proves the No-cloning theorem. □

4 Quantum money

Money has the flaw of being easily replicated. However, quantum states adhere to the no-cloning theorem, meaning an unknown quantum state cannot be duplicated. Initially, this might seem like a solution for using quantum states as currency, a concept proposed by Wiesner in 1983 (with an original manuscript dating back to circa 1969). However, unfortunately, his plan had its limitations, and devising a quantum money system without major drawbacks was significantly more challenging.

We will start by introducing the idea of public-key quantum money. Afterwards, we will look at the quantum money from knots scheme[FGH+12].

4.1 Public-Key quantum money

Several candidate public-key quantum money schemes have been proposed,

In 1983, Wiesner and others presented the idea of public-key quantum money [BBBW83]. Numerous quantum money protocols have been proposed since Wiesner's work, but only Farhi et al.'s knot-theoretic scheme has not yet been broken. A survey of efforts towards public-key quantum money can be found in section 9 of [Aar16].

The public-key quantum money scheme must satisfy three properties:

- The mint can produce money efficiently. There is a polynomial-time quantum generation algorithm that randomly produces a quantum money state $|\$_p\rangle$ and an associated serial number p. A list of all the valid serial numbers is published.
- Anyone with a quantum computer can efficiently verify that a quantum money state was produced by the mint without destroying the money. There is a polynomial-time quantum verification algorithm

$$\text{Ver}\colon (|\$_p\rangle, p) \overset{?}{\mapsto} (|\$^0\rangle, b),$$

  which, when given a valid money state, outputs an 'accepted/rejected' bit and, if accepted, the original money state.
- Given a valid quantum money state, nobody can copy it. There does not exist a polynomial-time quantum algorithm that takes as input $(|\$_p\rangle, p)$ and outputs two quantum money states such that the verification algorithm

accepts both quantum money states (when paired with p) as genuine with more than exponentially small probability.

4.2 Farhi et al.'s quantum money scheme

A quantum money scheme was introduced by Farhi et al. in 2010, which utilised superpositions of diagrams encoding oriented links with identical Alexander polynomials. The authors anticipate that the scheme will remain secure against adversaries with computational limitations.

The scheme's security is based on the assumption that it is difficult to determine whether two links are equivalent, even in the average case for a quantum computer. This problem is referred to as the recognition problem, and even the unknot recognition problem (determining if a given knot is equivalent to the unknot) has no known polynomial-time algorithm. The best-known algorithms for both problems have an exponential running time [BO14], which further supports the use of links as the basis for the scheme's security.

We will discuss how polynomial-time quantum money generation and verification work in this scheme. We will go through all the essential details but make some simplifications. For example, we will assume all superpositions discussed in the next section are uniform. In Farhi et al.'s paper, the quantum money states are not uniform superpositions; Instead, grid diagrams are weighted in the superposition based on the size of the corresponding permutations, according to a Gaussian distribution. This is in order to make specific attacks on the scheme less likely.

4.2.1 Generation

The generation of money in this scheme works as follows. The mint begins by producing superposition over all pairs of permutations on d elements for all $d \quad D$ for some large integer D:

$$\sum_{d=2}^{D} \sum_{\pi_X,\pi_O 2 S_d} j\pi_X, \pi_O i.$$

Next, the mint measures if π_X and π_O are disjoint or not (they are disjoint with probability close to $\frac{1}{e}$). If this is not the case, the mint starts over. If they are disjoint, the mint will be left with a superposition of all planar grid diagrams of size at most $D \quad D$. After normalising, we get

$$\frac{1}{\sqrt{N}} \sum_{\text{Grid diagrams } G} jGi.$$

Here N is the number of such diagrams. From this state, the mint computes the Alexander polynomial of G into a second register.

$$\frac{1}{\sqrt{N}} \sum_{\text{Grid diagrams } G} |G\rangle|A(G)\rangle.$$

The second register is measured, obtaining a polynomial p. The result is a state $|\$_p\rangle$ the weighted superposition of all planar grid diagrams with Alexander polynomial p and size at most $D \quad D$:

$$\frac{1}{\sqrt{N'}} \sum_{G \text{ with } _{G}=p} |G\rangle|p\rangle.$$

The two registers are no longer entangled. Separating them yields the money state

$$|\$_p\rangle = \frac{1}{\sqrt{N'}} \sum_{G \text{ with } _{G}=p} |G\rangle.$$

All these steps can be done in polynomial time. In particular, a polynomial-time classical algorithm for the Alexander polynomial can be converted into a polynomial-time quantum algorithm that works on superpositions of links.

4.2.2 Verification

Given a supposed valid piece of quantum money $(|\psi\rangle, p)$, the verifier needs to check whether $|\psi\rangle$ is, in fact, the proper superposition over all diagrams with Alexander polynomial p. We present the critical components of the verification scheme.

To verify quantum money, a quantum verification procedure based on a classical Markov chain is applied. Essentially, many randomly chosen Reidemeister moves are applied to the money state superposition with the restriction that Reidemeister moves that would expand the planar grid diagram beyond d dimensions are ignored.

The three Reidemeister, or more precisely, in the case of grids, the three types of Cromwell moves are sufficient to generate all planar grid diagrams of the same link. Consider S as the set of all possible Cromwell moves on planar grid diagrams of size d. Any move maps a grid diagram to another one representing a link of the same equivalence class. While each move $s \; 2 \; S$ has an inverse, it is, unfortunately, impossible to model this as a group action because not every move can be applied to any diagram.

Algorithm 1 Verifying Quantum Money

1. Verify that p is a valid published serial number.

2. Verify that $|\psi\rangle$ is a superposition over valid grid diagrams, i.e. two disjoint permutations. Let A be a quantum algorithm for this purpose. The verifier measures whether A accepts the input $|\psi\rangle$. If it does not, the money is invalid; otherwise, move on to Step 3. Note that if A accepts, the original state $|\psi\rangle$ may have included some invalid diagrams in the superposition (but we happened to measure a valid one). In any case, the new post-measurement state $|\psi'\rangle$ will only include valid diagrams.

3. Verify that $|\psi\rangle$ is a superposition over valid grid diagrams with polynomial p. The Alexander polynomial (of a superposition of links) is computed on the input $|\psi'\rangle$. The result is measured. If the result is p, move on to Step 2. The new state is some $|\psi''\rangle$ that is a superposition over diagrams with Alexander polynomial p. If the result is not p, the money is invalid.

4. Verify that the superposition over diagrams with Alexander polynomial p contains *all* such diagrams. This will be done by supplying Algorithm 2 with $|\psi''\rangle$.

We can, however, consider the Markov chain defined on grid diagrams with the update rule 'apply a random valid move'. For each link L, the set of all diagrams representing L is a stationary distribution for this Markov chain. Consequently, the set of links with Alexander polynomial p forms a stationary distribution as well. This means the diagrams in a valid quantum money superposition $\sum_i |p_i\rangle$ form a stationary distribution.

This Markov chain verification step, see Algorithm 2, will use slightly more quantum computing theory than introduced in this text. The main idea is to apply Cromwell moves. The money state will be invariant under these moves if it is valid.

4.2.3 Security

Why are links used in this quantum money scheme instead of other mathematical objects like graphs?

The reason is that links possess specific desirable properties. Firstly, there are numerous distinct links, and each link has many link diagrams encoding it. It has long been conjectured that the link equivalence problem, in other words, distinguishing whether or not two links are equivalent or not, is hard. It is not known if this problem is in NP. Even verifying whether some random knot is trivial is a non-trivial task! However, this problem was recently shown to be in NP, along with a quasipolynomial algorithm [Lac21]. If this could be generalised to other knot equivalence, it would spell trouble for the scheme's security.

Also, links have many invariants, such as the Alexander polynomial, which is also simple to calculate. The Alexander polynomial is chosen because, in contrast, other invariants like the Jones polynomial may be challenging to compute, even for quantum computers.

On the other hand, using graphs where equivalence is based on graph isomorphisms may not be ideal due to the solvability of the graph isomorphism problem on most random graphs in practice. In [Bab16], it is shown that graph isomorphism is solvable in quasipolynomial time.

Algorithm 2 Markov chain verification algorithm

1. Let S be the set of possible Cromwell moves on grid diagrams of dimension D. Each move $s \in S$ can be represented by a permutation matrix P_s that encodes the action of s on all grid diagrams of dimension D. In the case that the move s is an invalid move for a specific diagram, it will act as the identity. Let

$$V = \sum_{s \in S} P_s \otimes |s\rangle\langle s|$$

where the second register has basis states $|s\rangle$ for $s \in S$. Since permutation matrices are unitary, so is V.

2. Adjoin a second register to $|\psi^{00}\rangle$ and initialise it to the uniform superposition on basis elements $|s\rangle$ for $s \in S$.

$$|\phi\rangle = |\psi^{00}\rangle \frac{1}{\sqrt{|S|}} \sum_{s \in S} |s\rangle$$

3. Apply V to $|\phi\rangle$.

$$\begin{aligned} V|\phi\rangle &= \sum_{s \in S} P_s \otimes |s\rangle\langle s| \left(|\psi^{00}\rangle \frac{1}{\sqrt{|S|}} \sum_{s \in S} |s\rangle \right) \\ &= \sum_{s \in S} \frac{1}{\sqrt{|S|}} (P_s |\psi^{00}\rangle) \otimes |s\rangle \end{aligned}$$

If $|\psi^{00}\rangle$ is a valid money state, then $P_s|\psi^{00}\rangle = |\psi^{00}\rangle$, so $V|\phi\rangle = |\phi\rangle$, and the second register will be separable from the first, with the value $\sqrt{\frac{1}{|S|}} \sum_{s \in S} |s\rangle$.

Measure whether the third register of $V|\phi\rangle$ contains a $+1$ eigenvector of $\sum_{s,s' \in S} \frac{1}{|S|} |s\rangle\langle s'|$. If the result is no, then the money state is invalid. If the result is yes, repeat step 3 with the current state of the two registers instead of $|\phi\rangle$. The verification algorithm is complete once this step has passed a polynomial amount in D times.

5 The challenges of implementation

This section discusses some of the challenges of implementing quantum money from knots in a quantum simulator. Qiskit is an open-source framework developed by IBM for working with quantum computers. In Qiskit, users can create quantum circuits using quantum gates to manipulate qubits. It also offers tools for visualising and analysing quantum computations. Qiskit provides access to real quantum devices and simulators for running quantum circuits. For our exposition, we would like to inspect our quantum state, and for this purpose, the 'statevector simulator' backend is most appropriate.

In Qiskit, creating a uniform superposition over n qubits for some small integer n (say $n = 20$) is relatively simple. Such a superposition consists of 2^n basis elements, each encoded by a bitstring of length n.

Our first challenge is to create a superposition over planar grid diagrams, so we need a way to relate grid diagrams to bit strings. As discussed, a planar grid diagram can be represented by two disjoint permutations. A permutation of length n can be represented by a number from 1 to $n!$ By this reasoning, in an n qubit system, we can extract two bitstrings of length $k = \lfloor \frac{n}{2} \rfloor$. Now let m be the greatest positive integer such that $m!$ 2^k 1. This way, every basis vector represents a pair of permutations in S_m S_m, and every such pair is represented. In particular, we can create superpositions over pairs of disjoint permutations by filtering out the undesired basis vectors. This can be done through measurement or, because we are simulating, by simply looping over the basis vectors.

The next challenge is to create a superposition over planar grid diagrams with the same Alexander polynomial. Once we have a superposition of grid diagrams, the next step is to calculate the Alexander polynomial for each of these diagrams. Doing this sequentially would be very inefficient, as the number of diagrams grows exponentially with the number of qubits. Fortunately, we can convert a polynomial-time classical algorithm for calculating the Alexander polynomial to a polynomial-time quantum algorithm that takes a superposition of links as input and outputs a superposition of polynomials. Unfortunately, however, there is no straightforward way to implement such a conversion and doing this from scratch went beyond the project's scope. We are therefore forced to do the calculations sequentially.

Another big problem is that the currently available classical implementations of the Alexander polynomial, like the one in Sage, need to be more stable for our problems; the Sage function can often hang indefinitely and fail to return a result

for seemingly random links. The Python library pyknotid [ToSc17] also provides ways to calculate the polynomial, but its incompatibility with newer Python 3 versions makes it difficult to work with. Calculating the Alexander polynomial is, therefore, currently the biggest obstacle.

The next challenge is the verification of quantum money states. Assuming we could calculate Alexander polynomial efficiently, the first three steps of algorithm 1 would be easy. Implementing the Markov chain verification algorithm, however, while possible, would be quite the challenge. The main challenge is that we would need to encode each possible grid move and create a quantum system with these grid moves as basis states.

Another issue that might appear is quantum error and noise. Realistic quantum computers and quantum simulators are affected by noise and errors, which could degrade the security of quantum money as the scale increases. Error correction and noise reduction techniques need to be incorporated, which add to the complexity of the problem.

6 Conclusions and future work

This work explored the exciting domain of quantum money, shedding light on an innovative application of quantum mechanics and knot theory. We focused on understanding the quantum money scheme proposed by Farhi et al., which employs the mathematical intricacies of links, specifically the Alexander polynomial, to create secure and unforgeable quantum money.

In this endeavour, we have developed an understanding of the basics of knot theory, emphasising the Alexander polynomial. We delved into the fundamental principles of quantum mechanics to understand the workings of public-key quantum money. This led us to unpack the concept of public-key quantum money based on knots, exploring the generation and verification processes. Building on our exploration, we propose future work to focus on an efficient quantum algorithm for the Alexander polynomial and a better understanding of the verification procedure.

Despite the challenges, the potential of quantum money from knots is immense. Further research and development in quantum algorithms, quantum error correction, and quantum hardware will lead to efficient and practical implementations of this scheme in the future.

References

[Aar16] Scott Aaronson. The complexity of quantum states and transformations: from quantum money to black holes. *arXiv preprint arXiv:1607.05256*, 2016.

[AC12] Scott Aaronson and Paul Christiano. Quantum money from hidden subspaces. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 41–60, 2012.

[Bab16] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697, 2016.

[BBBW83] Charles H Bennett, Gilles Brassard, Seth Breidbart, and Stephen Wiesner. Quantum cryptography, or unforgeable subway tokens. In *Advances in cryptology: Proceedings of Crypto 82*, pages 267–275. Springer, 1983.

[BO14] Benjamin A. Burton and Melih Ozlen. A fast branching algorithm for unknot recognition with experimental polynomial-time behaviour, 2014.

[Bru21] P. Bruin. Syllabus topologie, versie van najaar 2021.

[BZH13] Gerhard Burde, Heiner Zieschang, and Michael Heusener. *Knots*, volume 5. Walter de Gruyter, 2013.

[CF12] R.H. Crowell and R.H. Fox. *Introduction to Knot Theory*. Graduate Texts in Mathematics. Springer New York, 2012.

[CL14] Alexander Coward and Marc Lackenby. An upper bound on Reidemeister moves. *American Journal of Mathematics*, 136(4):1023–1066, 2014.

[Cro95] Peter R Cromwell. Embedding knots and links in an open book i: Basic properties. *Topology and its Applications*, 64(1):37–58, 1995.

[Dyn06] Ivan Dynnikov. Arc-presentations of links: monotonic simplification. *Fundamenta Mathematicae*, 1(190):29–76, 2006.

[FGH+12] Edward Farhi, David Gosset, Avinatan Hassidim, Andrew Lutomirski, and Peter Shor. Quantum money from knots. In *Proceedings of the*

3rd Innovations in Theoretical Computer Science Conference, pages 276–289, 2012.

[Kau05] Louis H Kauffmans. Knot diagrammatics. *Handbook of knot theory*, pages 233–318, Elsevier, 2005.

[Lac21] Marc Lackenby. The efficient certification of knottedness and thurston norm. *Advances in Mathematics*, 387:107796, 2021.

[LAF+09] Andrew Lutomirski, Scott Aaronson, Edward Farhi, David Gosset, Avinatan Hassidim, Jonathan Kelner, and Peter Shor. Breaking and making quantum money: toward a new quantum cryptographic protocol. *arXiv preprint arXiv:0912.3825*, 2009.

[Lic12] WB Raymond Lickorish. *An introduction to knot theory*, volume 175. Springer Science & Business Media, 2012.

[Liv93] Charles Livingston. *Knot theory*, volume 24. Cambridge University Press, 1993.

[MK96] Kunio Murasugi and Bohdan Kurpita. *Knot theory and its applications*. Springer, 1996.

[Rol03] Dale Rolfsen. *Knots and links*, volume 346. American Mathematical Soc., 2003.

[RSdJ22] S.van der Lugt R. S. de Jong. Syllabus behorend bij het vak inleiding in de algebraïsche topologie, Versie 16 augustus 2022.

[Sto] Jasper Stokman. Quantum groups and knot knot theory, https://staff.fnwi.uva.nl/j.v.stokman/week37.pdf.

[ToSc17] Alexander J Taylor and other SPOCK contributors. pyknotid knot identification toolkit. https://github.com/SPOCKnots/pyknotid, 2017. Accessed 2023-30-06.

A Python code

The following Jupyter notebook can be found on https://github.com/Rubenschuit/quantum-money.

Quantum Money

June 27, 2023

1 Implementing quantum money from knots

In this notebook we will make an attempt to implement quantum money from knots, see https://arxiv.org/pdf/1004.5127.pdf. To start, we give an example of how to set up a uniform superposition in qiskit on $n = 4$ qubits.

[198]:
```
import numpy as np
import GridPyM as g
from scipy.sparse import csc_matrix
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, Aer,␣
↪transpile, assemble, execute
from qiskit.visualization import plot_histogram
from qiskit.circuit.library import ZGate, MCMT
from tqdm.notebook import tqdm
import random
import qiskit.tools.jupyter
```

1.1 Starting small: generate uniform superposition on 4 qubits

[199]:
```
import numpy as np
from qiskit import QuantumCircuit, Aer, execute

# Print the statevector in bra-ket notation with a maximum of 3 decimal places
def print_state(statevector, n):
    print("Amplitudes of the statevector: ")
    for i, amplitude in enumerate(np.asarray(statevector)):
        binary = f"{i:04b}"
        ket = "|" + binary.zfill(n) + ">"
        print(f"{ket}: {np.round(float(amplitude), 3)} ")

def test(n):
    # Create a quantum circuit with n qubits
    qc = QuantumCircuit(n,n)

    # Apply a Hadamard gate to each qubit to create a uniform superposition
    qc.h(range(n))

    # Execute the circuit on the local simulator
```

```
    backend = Aer.get_backend('statevector_simulator')
    job = execute(qc, backend)

    # Get the resulting statevector
    statevector = job.result().get_statevector(qc)


    # Filter out unwanted states, for example all even basis vectors
    for i in range(len(statevector)):
        if i % 2 == 0:  # replace with your condition
            np.asarray(statevector)[i] = 0

    print_state(statevector, n)

test(4)
```

```
Amplitudes of the statevector:
|0000>: 0.0
|0001>: 0.25
|0010>: 0.0
|0011>: 0.25
|0100>: 0.0
|0101>: 0.25
|0110>: 0.0
|0111>: 0.25
|1000>: 0.0
|1001>: 0.25
|1010>: 0.0
|1011>: 0.25
|1100>: 0.0
|1101>: 0.25
|1110>: 0.0
|1111>: 0.25
```

Our goal will be to create a superposition over grid diagrams. So we will need to find a way correspond grid diagrams to bitstrings.

1.2 From permutation to binary string

A permutation of n elements can be represented as a binary string of length n! using a technique called factorial number system or factoradic. In this system, each digit of the binary string represents a factorial base $(0!, 1!, 2!, 3!, ...)$, and the value of the permutation is obtained by multiplying each digit by its corresponding factorial base and summing the results.

Here's an example of how to represent the permutation $[2, 0, 1]$ as a binary string using the factorial number system:

First, we need to find the Lehmer code of the permutation. This is done by counting the number of elements to the right of each element that are smaller than it. For the permutation $[2, 0, 1]$, the

Lehmer code is $[2, 0, 0]$ because there are two elements to the right of 2 that are smaller than it (0 and 1), and no elements to the right of 0 or 1 that are smaller than them.

Next, we convert the Lehmer code to a binary string using the factorial number system. The first digit of the Lehmer code (2) is multiplied by its corresponding factorial base (2!) to obtain 4. The second digit (0) is multiplied by its corresponding factorial base (1!) to obtain 0. The third digit (0) is multiplied by its corresponding factorial base (0!) to obtain 0. The sum of these values is $4 + 0 + 0 = 4$.

Finally, we convert this value to a binary string to obtain the final representation of the permutation as a binary string. In this case, the binary representation of 4 is '100', so the permutation $[2, 0, 1]$ can be represented as a binary string '100'.

1.3 From binary string to permutation

First, we need to convert the binary string to its corresponding value in the factorial number system. In this case, the binary string '100' represents the value 4 in base 10.

Next, we need to convert this value to its corresponding Lehmer code using the factorial number system. To do this, we divide the value by the largest factorial base (2!) and obtain a quotient of 2 and a remainder of 0. The quotient becomes the first digit of the Lehmer code. We then divide the remainder by the next largest factorial base (1!) and obtain a quotient of 0 and a remainder of 0. The quotient becomes the second digit of the Lehmer code. We repeat this process until all digits of the Lehmer code have been obtained. In this case, the final Lehmer code is $[2, 0, 0]$.

Finally, we need to convert the Lehmer code to its corresponding permutation. To do this, we start with an ordered list of elements $[0, 1, 2]$ and use the Lehmer code to construct the permutation. The first digit of the Lehmer code (2) tells us that the first element of the permutation is the third smallest element in the list (2). We remove this element from the list and obtain $[0, 1]$. The second digit of the Lehmer code (0) tells us that the second element of the permutation is the first smallest element in the list (0). We remove this element from the list and obtain [1]. The third digit of the Lehmer code (0) tells us that the third element of the permutation is also the first smallest element in the list (1). We remove this element from the list and obtain an empty list. The final permutation is $[2, 0, 1]$.

[3]:
```python
from math import factorial

def binary_to_permutation(binary_string, n):
    # Convert the binary string to its corresponding value
    value = int(binary_string, 2)

    # Check if the value is within the valid range
    if not 0 <= value < factorial(n):
        raise ValueError

    # Convert the value to its corresponding Lehmer code
    lehmer_code = []
    for i in range(n - 1, -1, -1):
        quotient, value = divmod(value, factorial(i))
        lehmer_code.append(quotient)
```

```
    # Convert the Lehmer code to its corresponding permutation
    elements = list(range(n))
    permutation = []
    for code in lehmer_code:
        permutation.append(elements.pop(code))

    return permutation
```

[4]:
```
def permutation_to_factoradic(permutation):
    n = len(permutation)
    factoradic = [0] * n
    for i in range(n):
        factoradic[i] = permutation[i]
        for j in range(i):
            if permutation[j] < permutation[i]:
                factoradic[i] -= 1
    return factoradic

def factoradic_to_decimal(factoradic):
    n = len(factoradic)
    decimal = 0
    for i in range(n):
        decimal += factoradic[i] * factorial(n - i - 1)
    return decimal

def permutation_to_binary(permutation):
    factoradic = permutation_to_factoradic(permutation)
    decimal = factoradic_to_decimal(factoradic)
    binary = bin(decimal)[2:]
    return binary
```

[5]:
```
# Given an integer n, representing the number of bits available,
# return the largest integer m such that all permutations of
# length m can be reached by using n bits. This is the largest m
# such that m! < 2^n - 1.
def max_permutation_length(n):
    max_value = 2 ** n - 1
    m = 0
    while factorial(m) <= max_value:
        m += 1
    return m - 1
```

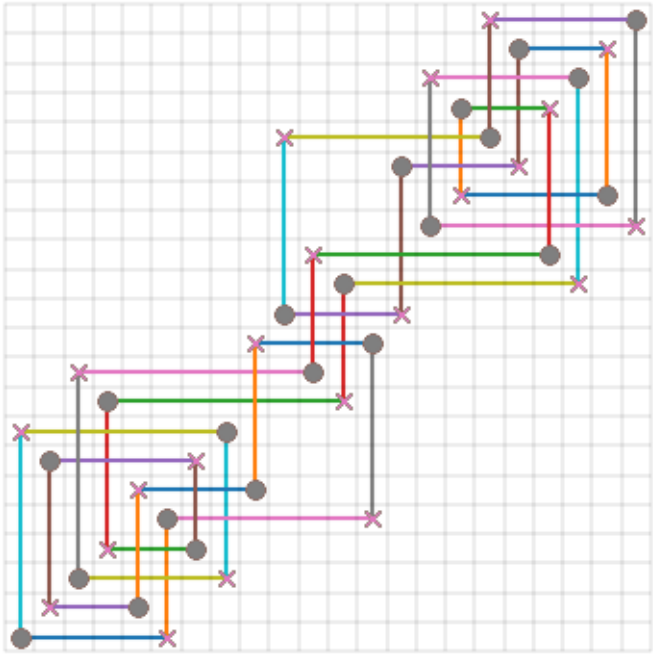
1.4 From two disjoint permutations to a planar grid diagram of a link

An $n \times n$ planar grid diagram is specified by two disjoint permutation of length n. GridPyM is a python library that allows us to work with disjoint permutation representations of links. See https://arxiv.org/pdf/2210.07399.pdf and https://github.com/agnesedaniele/GridPythonModule.

[7]:
```
# An invalid grid: the permutations are not disjoint. Gives output 1
G = [[0,1,2,3,4],[0,3,4,0,1]]
g.check_grid(G)
```

[7]: 1

[209]:
```
# Drawing example
G = [[5,1,7,3,12,4,6,0,11,2,8,13,19,10,21,15,17,9,18,14,20,16],
     [0,4,2,6,5,8,1,7,3,10,12,9,11,18,14,20,13,16,15,19,17,21] ]
if not g.check_grid(G):
    g.draw_grid(G, markings='XO')
else:
    print("invalid")
```



1.5 Gauss code

We use Sage for calculating Alexander polynomials. Unfortunately, Sage does not support the disjoint permutation representation of links. Sage does support oriented Gauss code.

Label the crossings from 1 to n (where n is the number of crossings) and start moving along the link. Trace every component of the link, by starting at a particular point on one component of the link and writing down each of the crossings that you encounter until returning to the starting point. The crossings are written with sign depending on whether we cross them as over or undercrossing. Each component is then represented as a list whose elements are the crossing numbers. A second

list of +1 and −1's keeps track of the orientation of each crossing.

[82]:
```
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Converts a grid consisting of two disjoint permutations (as lists) to␣
 ↪oriented gauss code supported by Sage.
def Gauss_code(grid):
    num_trivial_components=0
    A = grid[0]
    B = grid[1]
    final_gauss_code = []
    crossing_assignment = []
    c = 1
    coord = []
    columns = [i for i in range(len(A))]
    while len(columns) > 1:
        gauss_code = []
        column = min(columns)
        flag = False
        start = column
        while flag == 0:
            valueB = B[column]
            valueA = A[column]
            if valueB < valueA:
                for i in range(valueB + 1,valueA):
                    v1 = min(A.index(i), B.index(i))
                    v2 = max(A.index(i), B.index(i))
                    if v1 < column < v2:
                        pos = [column,i]
                        if pos not in coord:
                            coord.append(pos)
                            gauss_code.append(-c)
                            c = c + 1
                            if B.index(i) > A.index(i):
                                crossing_assignment.append(-1)
                            if B.index(i) < A.index(i):
                                crossing_assignment.append(1)
                        else:
                            caux = coord.index(pos) + 1
                            gauss_code.append(-caux)
            if valueB > valueA:
                for i in range(valueB - 1, valueA, -1):
                    v1 = min(A.index(i), B.index(i))
                    v2 = max(A.index(i), B.index(i))
                    if v1 < column < v2:
                        pos = [column,i]
                        if pos not in coord:
                            coord.append(pos)
```

```
                        gauss_code.append(-c)
                        c = c + 1
                        if B.index(i) > A.index(i):
                            crossing_assignment.append(1)
                        if B.index(i) < A.index(i):
                            crossing_assignment.append(-1)
                    else:
                        caux = coord.index(pos) + 1
                        gauss_code.append(-caux)
        if B.index(valueA) > column:
            for i in range(column + 1, B.index(valueA)):
                if min(A[i], B[i]) < valueA < max(A[i], B[i]):
                    pos = [i,valueA]
                    if pos not in coord:
                        coord.append(pos)
                        gauss_code.append(c)
                        c = c + 1
                        if B[i] > A[i]:
                            crossing_assignment.append(1)
                        if B[i] < A[i]:
                            crossing_assignment.append(-1)
                    else:
                        caux = coord.index(pos)+1
                        gauss_code.append(caux)
        if B.index(valueA) < column:
            for i in range(column - 1 ,B.index(valueA), -1):
                if min(A[i], B[i]) < valueA < max(A[i], B[i]):
                    pos = [i, valueA]
                    if pos not in coord:
                        coord.append(pos)
                        gauss_code.append(c)
                        c = c + 1
                        if B[i] > A[i]:
                            crossing_assignment.append(-1)
                        if B[i] < A[i]:
                            crossing_assignment.append(1)
                    else:
                        caux = coord.index(pos) + 1
                        gauss_code.append(caux)
        columns.remove(column)
        column = B.index(valueA)
        if column == start:
            flag = True
    if len(gauss_code) > 0:
        final_gauss_code.append(gauss_code)
    else:
        num_trivial_components += 1
```

```
    return([final_gauss_code,crossing_assignment])
```

1.6 Calculating the Alexander polynomial

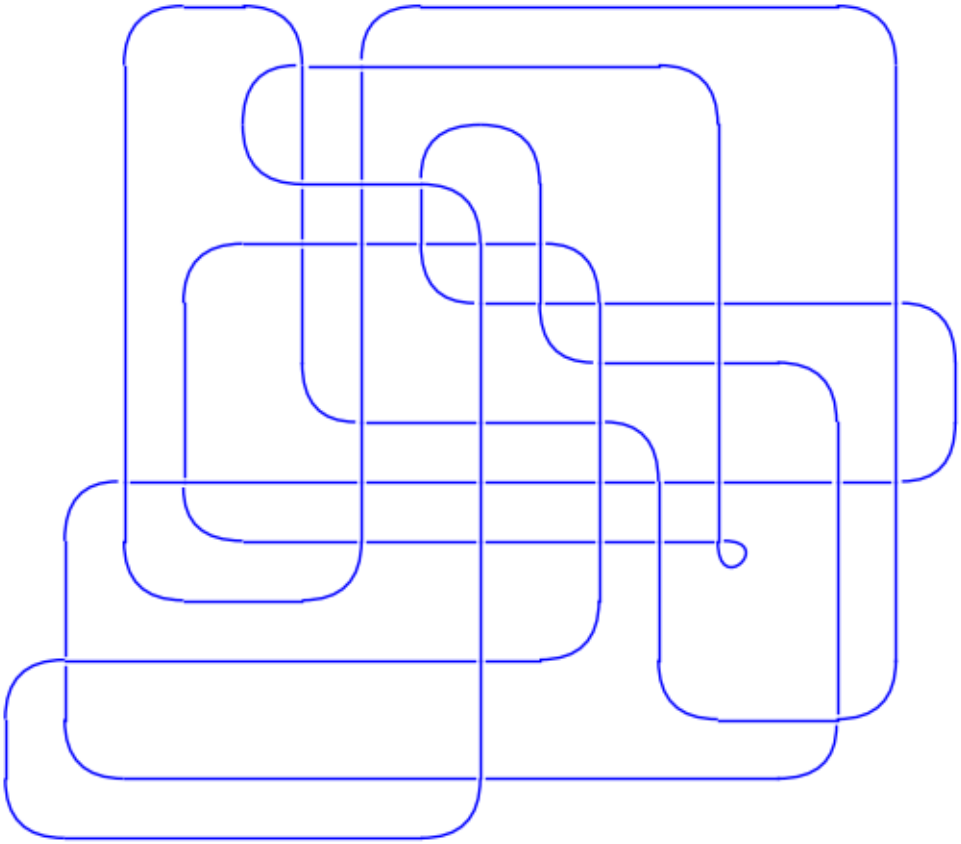
One of the few already existing implementations for calculating the Alexander polynomial of oriented links is found in the Sage library. Unfortunately, the function often hangs indefinitely.

[213]:
```
# Generate random 20 x 20 grid containing a 3-component link and plot it using␣
↪Sage
G = g.generate_random_grid(16, 3)
#g.draw_grid(G, markings='XO')
L= Link(Gauss_code(G))
L.alexander_polynomial()
```

[213]:
```
t^-1 - 2 + t
```

[196]:
```
# Generate random 20 x 20 grid containing a 3-component link and plot it using␣
↪Sage
G = g.generate_random_grid(20, 3)
#g.draw_grid(G, markings='XO')
L= Link(Gauss_code(G))
L.plot()
```

[196]:

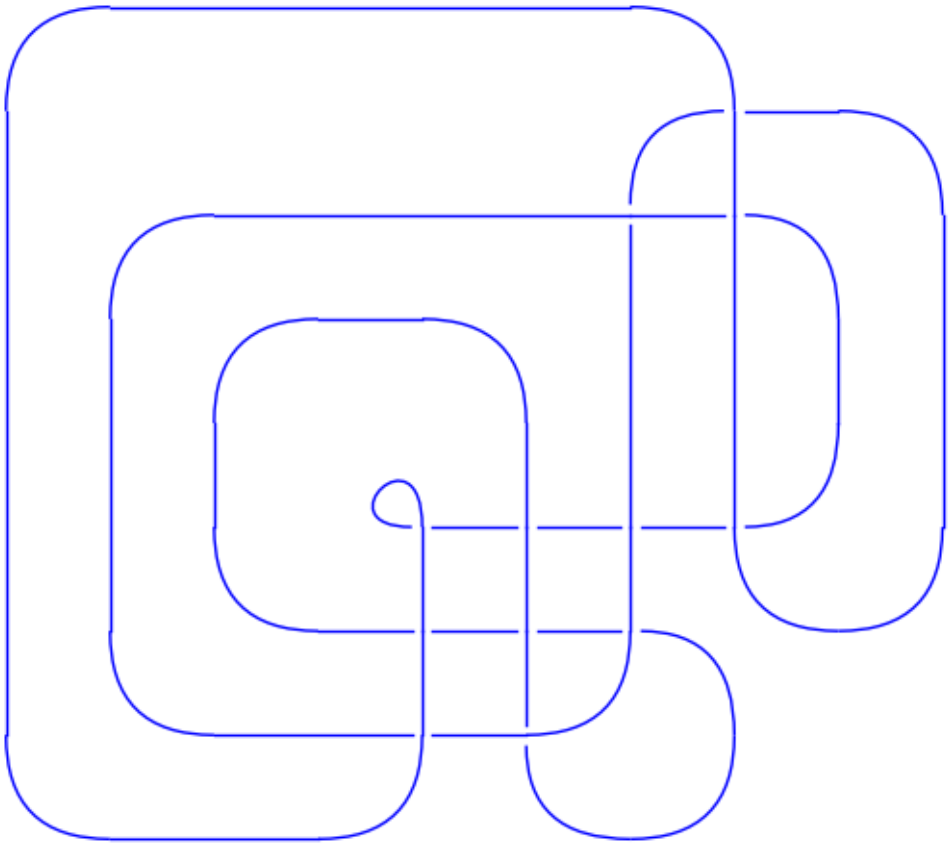


[197]:
```
# Calculate the Alexander polynomial, this often does not work for random links
↪on n x n grids with n > 18
L.alexander_polynomial()
```

[197]:
```
t^-2 - 4*t^-1 + 6 - 4*t + t^2
```

[214]:
```
# Sage doesnt like the following link for example
L = Link([[[1, 2, -2, -3, -4, -5, -6, 7, -8, 9, 10, 4, -7, -11, 5, 6, 11, 8],
↪[-9, -10, -12, -1, 3, 12]], [1, -1, 1, -1, -1, 1, -1, -1, -1, 1, -1, -1]])
#%prun L.alexander_polynomial() # Does not work for this L
L.plot()
```

[214]:



1.7 Generating a uniform superpositions of planar grid diagrams

[144]:
```
# Function to check if two permutations are disjoint
def is_disjoint(coord_perm1, coord_perm2):
    for i in range(len(coord_perm1)):
        if coord_perm1[i] == coord_perm2[i]:
            return False
    return True

# Prepare a uniform superposition on n qubits and filter it to a superposition␣
 ↪of grid diagrams
def prepare_state(n):
    # The largest m such that we can generate all permutations of length m with␣
 ↪floor(n/2) bits
    m = max_permutation_length(n // 2)

    # Create a quantum circuit with n qubits
    qc = QuantumCircuit(n,n)

    # Apply a Hadamard gate to each qubit to create a uniform superposition
    qc.h(range(n))

    # execute the circuit on the statevector simulator so we can inspect the␣
 ↪state afterwards
    backend = Aer.get_backend('statevector_simulator')
    result = execute(qc, backend).result()
    statevector = result.get_statevector()

    # Filter non disjoint permutations and unnecessary ones by setting their␣
 ↪amplitutes to 0
    for i in tqdm(range(len(statevector))):
        binary_string = bin(i)[2:].zfill(n)
        middle = len(binary_string) // 2
        first_half = binary_string[:middle]
        second_half = binary_string[-middle:]
        try:
            X = binary_to_permutation(first_half, m)
            O = binary_to_permutation(second_half, m)
            if not is_disjoint(X,O):
                np.asarray(statevector)[i] = 0
        except ValueError:
            # not all basis elements are required to generate all pairs of m␣
 ↪permutations
            np.asarray(statevector)[i] = 0

    # Normalise the result
    return statevector / np.linalg.norm(statevector)
```

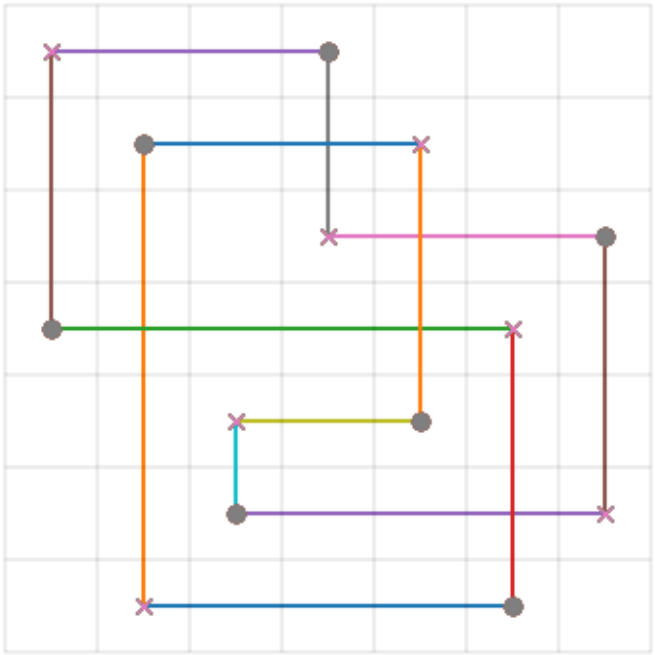
```
# L.alexander_polynomial() is too inconsistent unfortunately; it will hang often
def inspect_state(n, statevector, PRINT_RANDOM_SAMPLE = True, PRINT_DIAGRAM =␣
 ↪True,
                  PRINT_ALEXANDER = False):
    m = max_permutation_length(n // 2)

    # Convert to sparse representation as a lot of the amplitudes have been set␣
 ↪to 0
    sparse_statevector = csc_matrix(np.asarray(statevector))

    # Find indices of non-zero elements
    # looping over this list saves a lot of time compared to looping over the␣
 ↪statevector
    nonzero_indices = sparse_statevector.nonzero()

    sample = nonzero_indices[1]

    # Get random sample of grids
    if PRINT_RANDOM_SAMPLE:
        sample = random.sample(list(nonzero_indices[1]), 5)

    # Print some of the link diagrams generated
    for i in tqdm(sample):
        amplitude = statevector[int(i)]

        if np.isclose(0, amplitude):
            continue # skip basisvectors with 0 amplitude

        binary_string = bin(i)[2:].zfill(n)
        middle = len(binary_string) // 2
        first_half = binary_string[:middle]
        second_half = binary_string[-middle:]
        try:
            X = binary_to_permutation(first_half, m)
            O = binary_to_permutation(second_half, m)
            G = [X,O]
            if PRINT_DIAGRAM:
                g.draw_grid(G, markings='XO')
            if PRINT_ALEXANDER:
                L = Link(Gauss_code(G))
                print("Alexander polynomial: ", L.alexander_polynomial())
        except ValueError:
            pass
        except AssertionError as e:
            print(e)
```

[110]:
```
# Going higher then 30 qubits is not feasible
n=30
statevector = prepare_state(n)
```

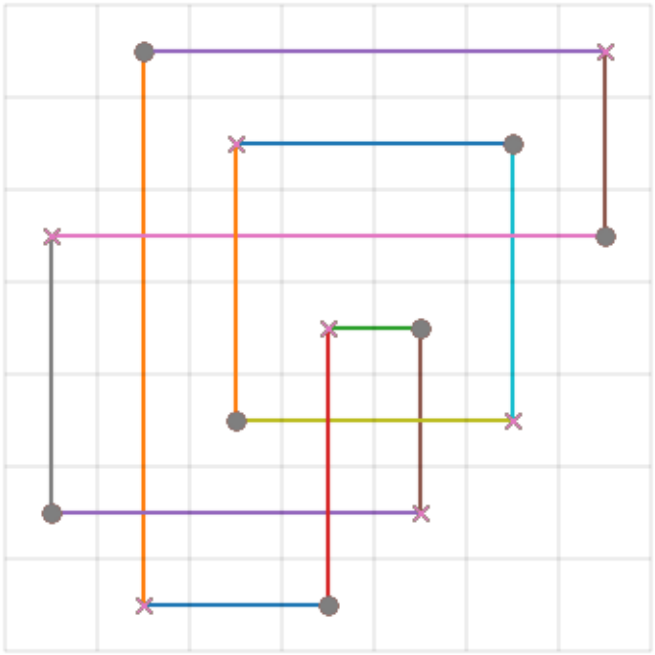
```
  0%|          | 0/1073741824 [00:00<?, ?it/s]
```

[146]:
```
inspect_state(n, statevector, PRINT_ALEXANDER=True)
```

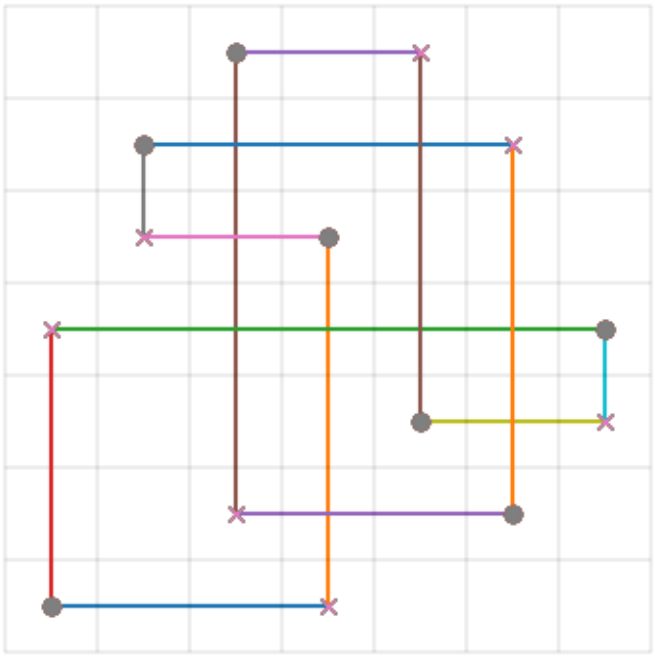
```
  0%|          | 0/5 [00:00<?, ?it/s]
 (X,O) = ( [1, 6, 2, 5, 3, 4, 0] ,  [5, 2, 4, 0, 6, 1, 3] )
```



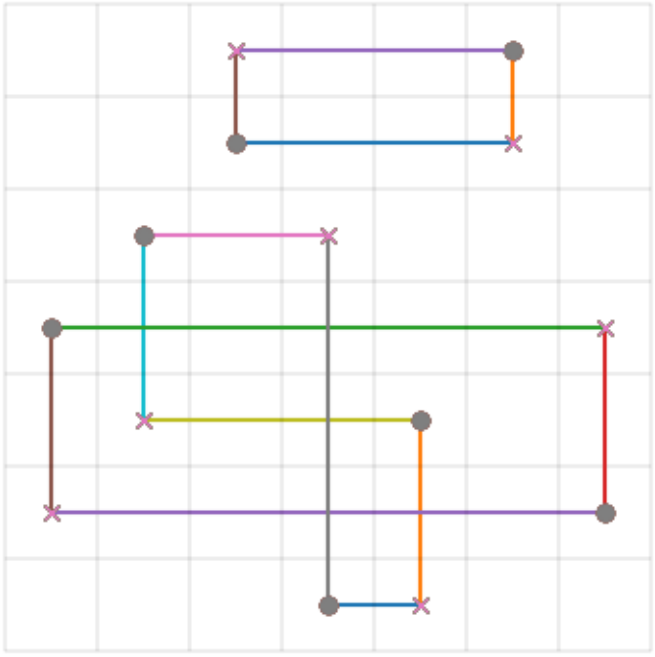
```
Alexander polynomial:  -t^-1 + 3 - t
 (X,O) = ( [1, 4, 5, 3, 0, 2, 6] ,  [3, 0, 2, 4, 6, 5, 1] )
```



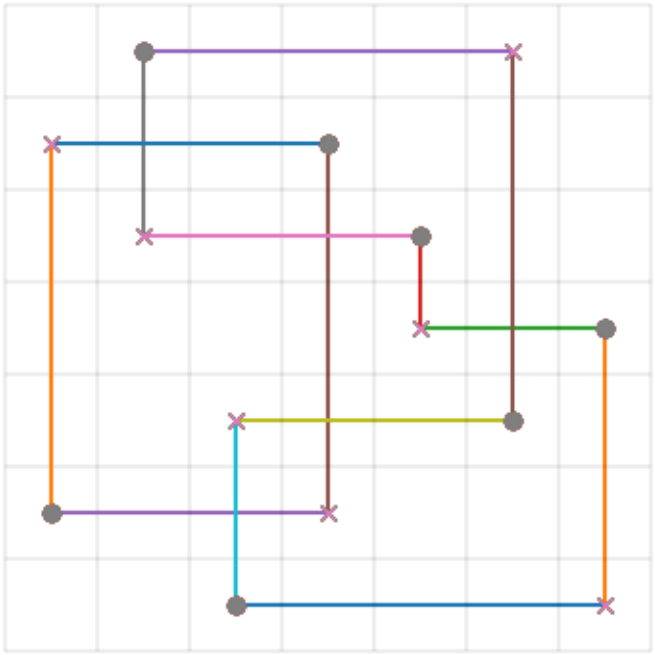
```
Alexander polynomial:  0
 (X,O) = ( [3, 2, 6, 0, 1, 5, 4] ,  [0, 5, 4, 6, 3, 1, 2] )
```



```
Alexander polynomial:  t^-1 - 1 + t
 (X,O) = ( [4, 0, 1, 6, 3, 5, 2] ,  [3, 6, 4, 0, 1, 2, 5] )
```



```
Alexander polynomial:  0
 (X,O) = ( [6, 3, 2, 4, 1, 0, 5] ,  [2, 0, 5, 6, 4, 3, 1] )
```



```
Alexander polynomial:  -2*t^-1 + 2
```