university of groningen

faculty of science
and engineering

mathematics and
applied mathematics

Observations on Almost OU Tangles

Bachelor’s Project Mathematics

September 2020

Student: Albert Silvans

First supervisor: Dr. Roland van der Veen

Second assessor: Dr. Alef Sterk

Contents

1 Introduction **4**

2 Tangles **5**
2.1 Tangle diagrams 6
2.2 Reidermeister moves 7
2.3 Oriented Gauss notation 7

3 Reiteration: OU tangle diagrams **8**
3.1 Why the OU form matters 10
3.2 The Glide move 10
3.3 Acyclic Diagrams 11
3.4 OU Algorithm 11
3.5 Issues arising with the approach 12

4 The "Scanning" tangle Drawing Algorithm **13**

5 Incidence matrices **14**

6 Conclusions and Further work **19**
6.1 What we did 19
6.2 Other things to consider 19
6.3 Convergence 19
6.4 Code and accumulated data 19

1 Introduction

In the article [Bar-Natan *et al.*, 2020] we are introduced with some Knot Theoretic structures including tangles, tangle diagrams, braids, most importantly the Over then Under form of a tangle, or OU form for short, followed by a proposed algorithm for bringing a tangle diagram into an OU form, and some results relating to this form.

Although the results presented in the previously mentioned article are quite broad and general, in our article for the sake of simplicity we restrict ourselves to the special case of a tangle with a single strand, and focus on what properties or patterns we can discover by examining 1-tangles under the process of the OU form algorithm. **In short: does the OU algorithm produce recognisable patterns, and if so what do they mean?** We shall see that our choice of tangles to study will prove to be erroneous, and that we will encounter a level of complexity that we did not bargain for. Specifically, the only tangle diagrams that behave nicely under the algorithm are trivial, and even then there is a set of trivial diagrams which do not behave nicely. Furthermore the algorithm in the case of misbehavior never terminates, and the diagrams that are produced at each iteration slowly tend to the realm of the wild.

How to read this article:

1. If the reader is unfamiliar with the basics of Knot Theory, then Section 2 is highly recommended, otherwise it may be skipped. For more Knot Theory we recommend [Adams, 1994] and [Lickorish, 1997].

2. If the original article [Bar-Natan *et al.*, 2020] has been read, then most of section 3 may be skipped with the exception of sections 3.4 and 3.5, since they outline differences in approach.

3. The rest of the article in order.

We set out in this article to probe the growing complexity of a sequence of tangle diagrams under a simple algorithm. We do this by drawing them, and by devising methods to symbolically represent them. The work falls short of making any concrete results or statements but possibly asks enough questions to catch the interest of the reader. The tangle diagrams do become increasingly complex, their corresponding incidence matrices have interesting repeating and nearly symmetric structures, and some tangle diagrams behave differently and some oddly similarly. A sizable amount of visual data has been accumulated and is freely available at [Silvans, 2020] for further discussions.

2 Tangles

Our main focus is on tangle diagrams, so we should define what they are sooner than later. The path to do so is rather winding: what tangle diagrams represent is best expressed first in 3 dimensions, as a construction in $\mathbb{R}^3$, but then projected in a nice way into the plane $\mathbb{R}^2$. Knot theory and its subdiscipline of studying properties of tangles is (considered by many) a topological question, though we will barely touch on any topological concepts besides the following few definitions. We will mainly view our diagrams as “beautified” graphs: points of connection, like vertices, and strands, like edges, except that we deeply care about the position and orientation of the vertices and edges.

Definition 1. (n-Tangle) Consider the closed unit ball B_3 in Euclidean 3-space, and its boundary the unit sphere $\mathbb{S}^2$. Take $n \in \mathbb{N}$ and for $1 \leq i \leq n$ take pairs of points $a_i, b_i \in \mathbb{S}^2$ such that $a_i \neq b_i$ for all $1 \leq i, j \leq n$.

For each $1 \leq i \leq n$ we have a map $\phi_i : [0, 1] \to B_3$ with the following properties:

1. $\phi_i(0) = a_i$ and $\phi_i(1) = b_i$,
2. $\phi_i((0, 1)) \subset B_3 \backslash \mathbb{S}^3$, i.e. the image of $(0, 1)$ is fully contained in the interior of B_3,
3. and for any other $1 \leq j \leq n$, $j \neq i$, we have $\phi_i([0, 1]) \cap \phi_j([0, 1]) = \varnothing$.

The closed ball, the pairs of points, and the maps connecting the points together form what we call an *n-tangle*.

We call the maps ϕ_i the *strands* of the tangle, and the pairs of points we call the *tips* of the strands. Each tangle inherits an orientation from the strands, specifically, from the parametrization ϕ_i, so $\phi_i(0)$ is the starting tip, and $\phi_i(1)$ is the ending tip of the i-th strand.

With this definition we have created a large collection of structures, however not all of these structures are notably different. Just like we know that a jumbled mass of cables, or perhaps, headphones by some pulling and repositioning is equivalent to untangled headphones, so here with n-tangles we have a similar notion of equivalence. We can sort these n-tangles into equivalent sets. We do this by Isotopy or ambient homeomorphisms.

Definition 2. (Isotopy/Ambient homeomorphisms) Two n-tangle diagrams T_1, T_2 are equivalent, if there exists a homeomorphism $h : B_3 \to B_3$ with $h(T_1) = T_2$ for which we can find a continuous family $h_t : B_3 \times [0, 1] \to B_3$ with $h_0 = \text{id}$, the identity homeomorphism, and $h_1 = h$.

The main difference of the intuitive example of the headphones, and this definition is that isotopy does not simply move the strands of the n-tangle, it also moves the rest of the closed ball B_3 with the strands. This process avoids contracting all the features of a tangle to a single point, and other anomalies of a similar nature. If we “pull” at the tips of the tangle, the lack of thickness of the strands allows for kinks and loops to vanish under regular homeomorphisms; something that is not possible with headphones, since they indeed have thickness (surely this would be the holy grail for any audiophile otherwise).

Finally, we need to address the question of the behaviour of tangles. Definition 1 does not rule out that one of the strands could be specified by a map like the topologist's sine curve, i.e. some pathological, misbehaving curve that makes all discussions difficult or at least tedious. We hence define *Tame* tangles and *Wild* tangles.

Definition 3. (Tame and Wild tangles) We call a tangle *tame*, if it is isotopic to a tangle the strands of which are finite polygonal chains, i.e. are piecewise linear with finitely many pieces. A tangle that is not tame we call *wild.*

This condition is not given in Definition 1 immediately for the sake of generality, although hard to study wild tangles pop up naturally in discussions, including our discussions later. However we never explicitly work with wild tangles.

2.1 Tangle diagrams

As humans we understand tangles quite well as a construction in $\mathbb{R}^3$, however it is often quite hard to work with such notions, in recent years there have been developments in AR (Augmented reality) devices which help model and visualise 3 dimensional structures, however for most of Math. history we have been limited with pencil and paper. Hence we draw tangle diagrams.

Intuitively a tangle diagram is just that: a diagram. One finds a clear area on their paper, draws some lines with some over and under strands in mind. Typically an under strand is denoted by a break in the strand where the crossing should be, we see an example of a tangle diagram in Figure 1. We may contruct a tangle diagram from a particularly nice tangle.

Definition 4. (Tangle diagrams) Let T be a tangle with all tips of the strands on the same great circle of $\mathbb{S}^2$, and proceed as follows:

1. orient the great circle parallel to a plane,

2. project the strands and great circle of the tangle orthogonally to the plane,

3. if necessary apply isotopies to T so that in the projection no more than two strands intersect at a point, and one strand self intersects at most once at a point.

4. Finally, indicate in the projection at each crossing which strand was over and which was under.

We refer to this construction as the diagram D of the tangle T.

We consider further primarily tame tangles, and tangles of a single strand. As well we focus on tangle diagram and mostly take for granted the topological considerations used to define tangles. We will refer to tame 1-tangle diagrams simply as tangle diagrams.

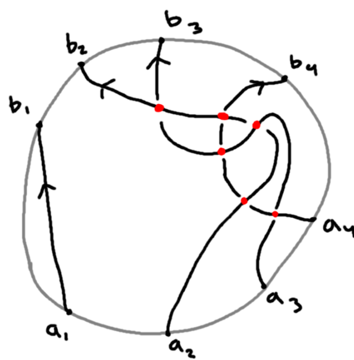
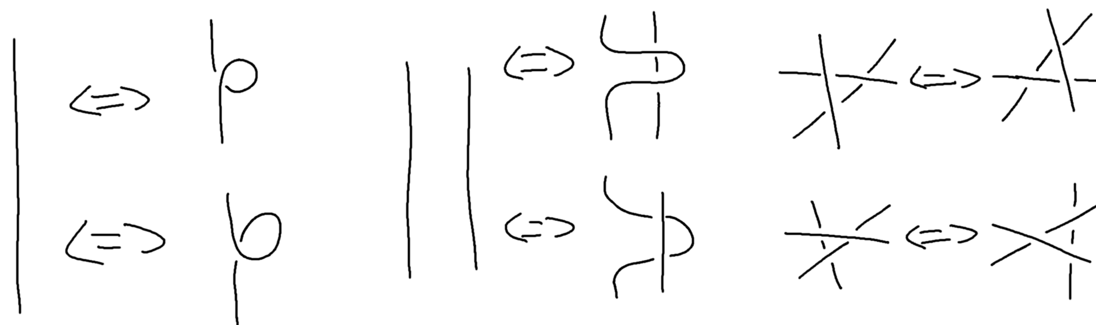


Figure 1: Example of a tangle diagram with 8 labeled points on the boundary (grey), each crossing (red). The orientation of a strand is indicated by an arrow.

2.2 Reidermeister moves

An exceedingly useful theorem that aids discussing tangle diagrams is the Reidermeister moves theorem. The theorem relates isotopy of tangles to equivalent changes in a tangle diagram. With this tool we may abstract even further, and really only worry about the number and position of crossings in a diagram.

Definition 5. (Reidermeister moves/ Isotopies) Two tangle diagrams are equivalent, if there exists a finite sequence of moves as in Figure 2 that brings one diagram into the shape of the other.



(a) The first move (b) The second move (c) The third move

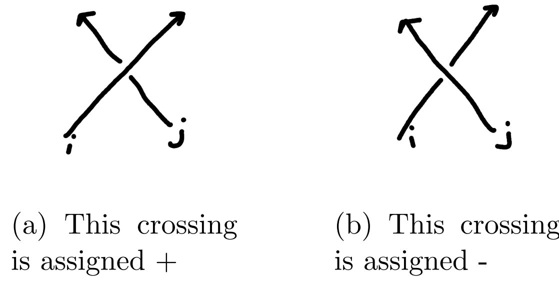
Figure 2: The 3 basic Reidermeister moves

2.3 Oriented Gauss notation

As mentioned before we focus on the combinatorial information of a tangle diagram, so we ignore the parametrizations of each strand, and only care about the crossings and how they appear on the strands. We may abstract our diagrams to hold only this information, to do so we chose the Oriented Gauss notation or Oriented Gauss code. To produce the Oriented Gauss code we follow the steps:

1. Prescribe an order to the strands

2. For each strand prescribe an orientation, this is typically given by the map ϕ_i associated to the strand in question,
3. For each crossing assign a name, typically a unique integer, and a sign either $+$ or $-$ taking into account the orientations of the strands as shown in Figure 3,
4. For each strand associate a string of symbols by following from the start point to the end point and for each crossing we encounter we record either the symbol O_i^s or U_i^s where O for over if our current strand is over, U for under, s for the sign of the crossing, and i the name of the crossing encountered.
5. The ordered set of the previously recorded strings is called the Oriented Gauss code.



(a) This crossing is assigned $+$

(b) This crossing is assigned -

Figure 3: Types of crossings by orientation.

From any tangle diagram we can produce a Gauss code, however we do not get a proper tangle diagram from any Gauss code. We provide some (non) examples of tangles and their Oriented Gauss codes in Figure 4

3 Reiteration: OU tangle diagrams

Now we recall a few important concepts from the article [Bar-Natan *et al.*, 2020], on which we base our approach. We define an operation on tangle diagrams, an algorithm, and mention that for the set of diagrams we consider the algorithm does not produce a final result. Instead the algorithm approaches wild tangles.

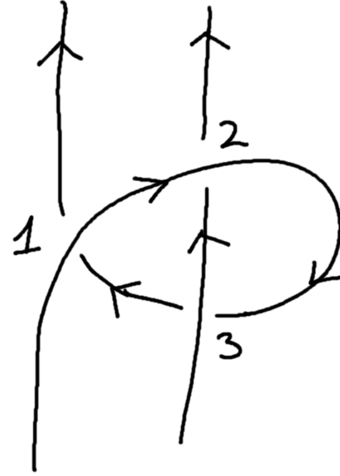
Using the Oriented Gauss code we define an OU tangle diagram.

Definition 6. An OU tangle diagram is a tangle diagram with an Oriented Gauss code, the strings of which can be split in two substrings where the first consists of all Os and the second consists of Us

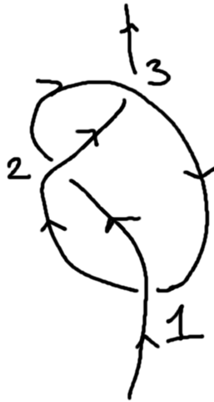
We find an example of an OU tangle diagram in Figure 4b, the first string can be split into the substrings $O_1^+O_2^+$ and $U_3^+U_1^+$, the second string can be split into substrings O_3^+ and U_2^+, hence the diagram is indeed OU. Likewise Figure 4a is also an OU tangle diagram, however Figure 4c is not, since the Os and Us are mixed together.

In generality for tangles with more than 1 strand, we need more tools to prove anything useful. However for 1-tangles, we can already make a straight forward observation.

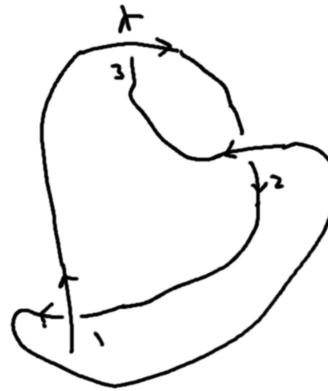
(a) $O_1^+U_1^+$



(b) $O_1^+O_2^+U_3^+U_1^+, O_3^+U_2^+$ where the left strand is first



(c) $O_1^+U_2^+O_3^+U_1^+O_2^+U_3^+$



(d) $O_1^+O_3^+U_2^+U_1^+O_2^+U_3^+$

Figure 4: Some (non) examples of Oriented Gauss codes

Theorem 1. All OU tangle diagrams of 1 strand are equivalent by Reidermeister moves to the trivial tangle diagram.

Proof. Consider an OU tangle diagram of 1 strand. Since it is OU, we can split the strand into two parts, one which consists of all over strands, and one with all under strands. Having split the strand into two parts, we notice that the parts are independent of one another, i.e. since the pieces are either all over or all under, they have no interconnections or flips due to crossings. Finally, we can reshape each piece independently to form a straight line, and in the process remove all the crossings. Hence we have a trivial tangle. ■

The question is then why do we work with 1-tangles, if they behave so poorly with regard to the *OU* property? We will shortly see that only a very specific subset of diagrams behave well.

3.1 Why the OU form matters

The initial goal of the *OU* form was to completely classify tangle diagrams. In fact the first assumption was that the *OU* form always existed and was unique. This certainly makes research more systematic because such a form would be *canonical*, i.e. a regular form for any diagram, one with which we could discern any two diagrams immediately simply by inspection.

This would in a way “solve” tangle diagrams since we would have a systematic approach to their study. However this was too ambitious, and the *OU* form does not always exist. In fact the set of diagrams of *OU* form is in correspondence with the subset of diagrams that are *braids*, of which there is extensive study and understanding.

So what about the tangles that do not fit into this picture? Here we try to find some patterns or structures arising from the Glide move and the OU algorithm in the tangles that do not behave well. We choose to work with 1-tangles, since they are the simplest (the least number of strands), so that we have less to take into account.

3.2 The Glide move

The Glide move as defined in [Bar-Natan *et al.*, 2020] is simply an alias for the composition of an R2 and R3 move. We illustrate the Glide move in Figure 5.

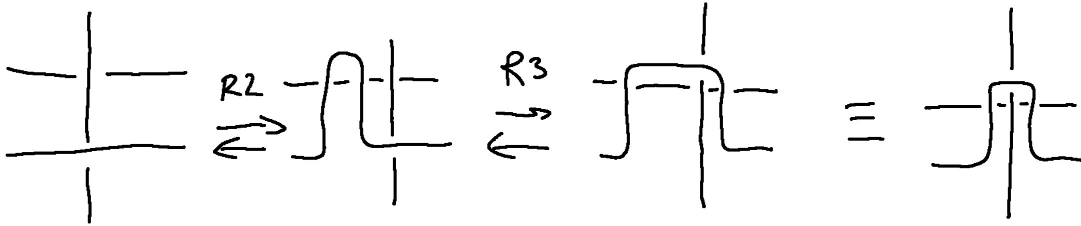


Figure 5: The Glide move is equivalent to an application of R2 and R3.

The goal of the Glide move is to swap positions of over strands and under strands, i.e. if the Gauss code of a tangle diagram contains $U_i^s O_j^t$ then by a glide move we can swap those symbols to produce a Gauss code with $O_i^t U_j^s$. However this is not the only change the Glide move introduces, other parts of the code are changed since we introduce two new crossings. We can translate the Glide move into an operation on the Oriented Gauss code. This makes it humanly possible to follow the evolution of the diagram upon Gliding. For a tangle diagram with crossings a, b we distinguish 4 cases:

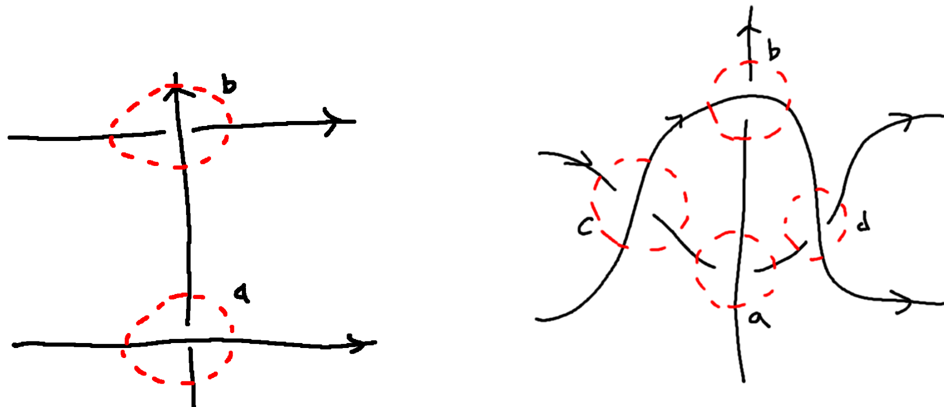
$$U_a^+ O_b^-, O_a^+, U_b^- \mapsto O_a^- U_b^+, O_c^- O_b^+ O_d^-, U_c^- U_a^- U_d^+$$

$$U_a^+ O_b^+, O_a^+, U_b^+ \mapsto O_a^+ U_b^+, O_c^+ O_b^+ O_d^+, U_d^- U_a^+ U_c^+$$

$$U_a^- O_b^-, O_a^-, U_b^- \mapsto O_a^- U_b^-, O_d^+ O_b^- O_c^+, U_c^+ U_a^- U_d^+$$

$$U_a^- O_b^+, O_a^-, U_b^+ \mapsto O_a^+ U_b^-, O_d^+ O_b^- O_c^-, U_d^+ U_a^+ U_c^-,$$

i.e. we replace the substrings of the current Gauss code on the left with their corresponding substrings on the right. We have that $a, b, c, d \in \mathbb{N}$, the indices c, d can be any number not yet used in the Gauss code. Each case corresponds to a combination of crossing signs, we illustrate the first case in Figure 6.



(a) Before applying the Glide move (b) After applying the Glide move

Figure 6: Example of the Glide move, the triple $U_a^+ O_b^-, O_a^+, U_b^-$ are each replaced with $O_a^- U_b^+, O_c^- O_b^+ O_d^-, U_c^- U_a^- U_d^+$

3.3 Acyclic Diagrams

Definition 7. (Cyclic Tangle Diagrams) A "cascade path" is a path on the tangle diagram that begins on one of the strands, follows the orientation of the strand it is on, and at a crossing may drop from the upper strand to the lower strand, if desired. A *closed* cascade path is one that returns to its starting position. A diagram is *cyclic* if there exists a closed cascade path, a diagram that is not cyclic is called *acyclic*.

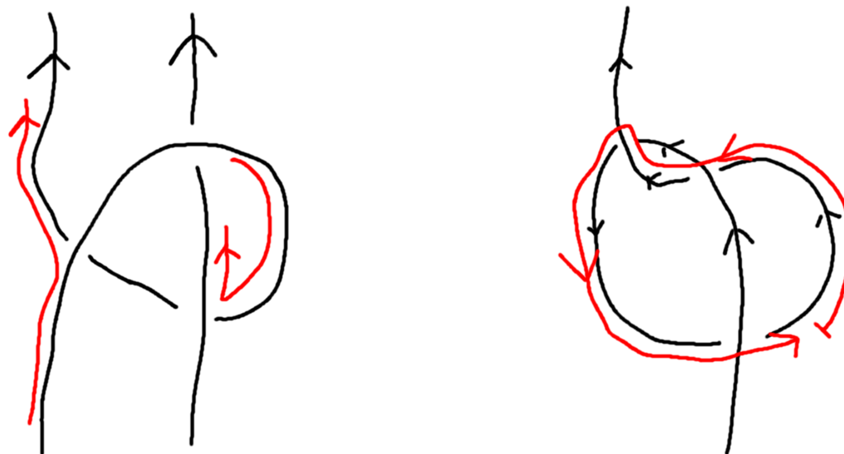
An example of cascade paths and cyclic paths is given in Figure 7. The importance of this concept of cyclicity will be made clear shortly, when we mention the OU algorithm and our variation on it.

3.4 OU Algorithm

We are now in the position to describe the algorithm from the paper:

1. For each string in order of occurrence in the Gauss code of the tangle, check each pair of consecutive symbols
2. If we encounter a pair $U_i^s O_j^t$, apply the Glide move,
3. If the diagram is in an OU form, we are done, if not return to step 1.

One will notice that the algorithm we outline here is not exactly as given in [Bar-Natan *et al.*, 2020], there after step 2 the diagram was simplified by R1 and R2 moves. We choose not to do this step, since it is one less moving part in the



(a) The left red path is cascade since we drop from an over strand to an understrand, the right path is not, since we go up.

(b) The trefoil with a cycle indicated in red. Notice the drop from the over strand to the under on the left crossing.

Figure 7: (Non) examples of cascade paths, and cyclic diagrams

mechanism. The simplifications are not easily tracked, and one might notice some miraculous simplifications completely changing the picture, removing "artifacts" in a way we cannot easily follow. Not reducing by R1 and R2 moves should make it simpler to conjecture statements.

For a tangle diagram D we write $\Gamma(D)$ to represent the diagram resulting from the application of the OU Algorithm on D. We would hope to see that for each diagram D there exists a $\Gamma(D)$, however it is shown in Theorem 2.3 of [Bar-Natan *et al.*, 2020] that $\Gamma(D)$ only exists for diagrams that are *acyclic*.

3.5 Issues arising with the approach

So if we encounter an OU 1-tangle diagram, there is nothing much to discuss, the distinction is not useful, since trivial tangle diagrams form only a small proportion of all tangle diagrams. The converse of the statement is not true however, a trivial tangle diagram can also be non OU, simply consider the tangle containing a single Reidermeister 1 move (refer to Figure 2a the bottom diagram with UO), and here the cyclic property helps distinguish these cases.

We had made the decision to not reduce by R1 and R2 moves, however this produces issues with the original implementation of the Glide move, specifically, there is a family of tangle diagrams for which the Glide move is not well defined. For example in Figure 8 the OU algorithm will not successfully make a glide move, since the move tries to swap the crossing with itself. As we push the crossing along the under strand, the over strand retreats equally, in such a way we simply circle around indefinitely. Any diagram that contains an R1 move like this will have issues under the OU algorithm, so we decide to ignore crossings like this in step 2 of the algorithm. This then implies that in some cases the algorithm can terminate erroneously, which is not a serious problem, since we can simply remove the R1 loop with no issues afterwards.

Figure 8: A very troublesome tangle

4 The "Scanning" tangle Drawing Algorithm

We present a 1-tangle drawing algorithm and a proof of its validity. We call it the "Scanning" tangle drawing algorithm because one can picture it as a process similar to a braiding board but upside down. The pieces of our finished diagram are attached to the braiding board, and we hold a set containing crossings that we still need to attach. We "scan" the diagram already on the braiding board and try to find a crossing that we can attach. The scanning must be precise and greedy, otherwise the method is not guaranteed to work. The generic steps of the algorithm can be seen in Figure 10. We now give the algorithm again but more precisely:

1. If there are no crossings, we are done, otherwise proceed.

2. Label each segment between the crossings of the tangle. Consider the set of crossings with each end labeled according to the respective connections in the tangle.

3. Take the first crossing that we encounter along the tangle, leave the end from which we came and attach the three other ends to the scanning front.

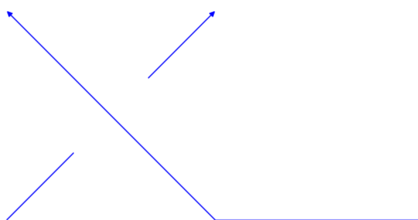


Figure 9: The first crossing, the left incoming strand we leave be, the right incoming strand we raise to the "scanning" front.

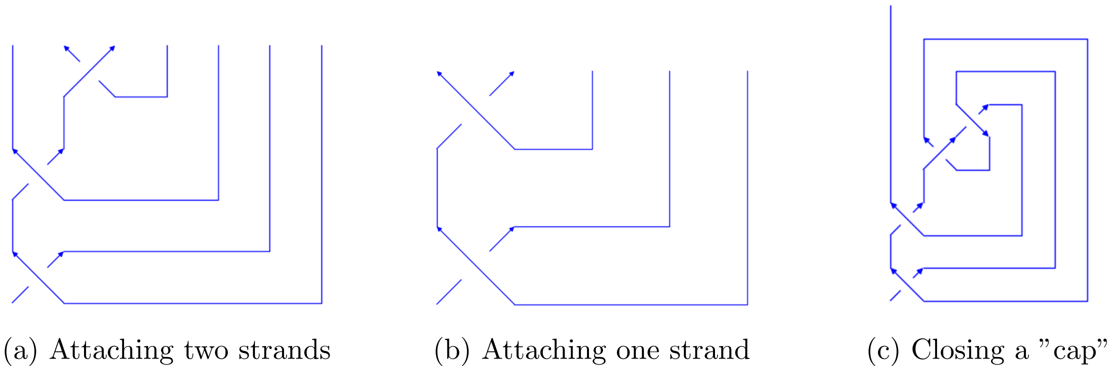
4. At each iteration read the labels of pairs of ends attached to the front from left to right, and attempt one of the following

   (a) If possible attach a new crossing to the two strands, if there exists an unused crossing with those labels (we allow rotating the crossings)

(b) If possible attach a new crossing to the left strand, if there exists an unused crossing with that label

(c) If neither are possible, try to "cap" off the strands, if they have the same label.

If none of this is possible with the pair, consider the next pair given by the right strand of the previous pair and the strand to the right of it. If none of the consecutive pairs work, then try (b) with the right-most strand.



(a) Attaching two strands (b) Attaching one strand (c) Closing a "cap"

Figure 10: What iterations of the scanning method look like

5. If there is only one strand left on the front, we are done, if there are more, repeat the previous step.

Proof. What our algorithm essentially does is provide a nice drawing of the tangle diagram, nice in the sense that it aims to not produce extra crossings in the process. It is given that the tangle diagram is planar by definition, so if at every step the diagram with scanning front is planar, then we are guaranteed to have a planar drawing, with no extra crossings.

We see that at the first step, when we fix the first crossing, the diagram is planar. Suppose at some iteration of the algorithm we have a planar diagram, then by affixing a crossing or capping off two strands we do not change the planarity of the diagram. Hence once the algorithm terminates, we have a planar tangle diagram.

Suppose at some iteration the algorithm produces a diagram with a cap that closes off a loose strand as in Figure 11. If this happens we can trace out a closed region as indicated in red. The crossing on the left lets a strand enter the region, but since the region is closed it cannot leave the region without introducing extra crossings, this implies that the original input for the algorithm did not encode a planar diagram to begin with. So we need not worry about this situation occuring ∎

5 Incidence matrices

Tangle diagrams essentially represent a combination of connections, i.e. strands connect crossings, like edges connect vertices on a graph. Given this similarity, we can construct incidence matrices for the crossings of our tangle diagrams.

We construct an incidence matrix for a tangle diagram as follows:

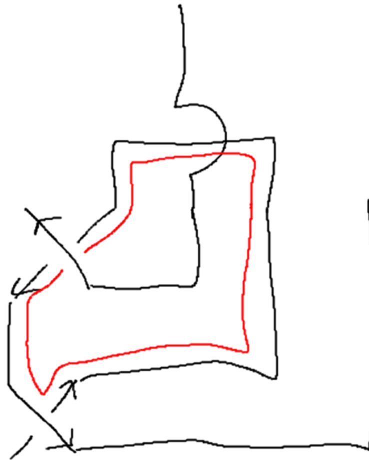


Figure 11: A supposed iteration of the scanning algorithm: we cap off a strand that still needs to be connected. This cannot happen, since the region indicated in red is a closed loop, this implies the input to the algorithm was not planar at the start.

1. Enumerate crossings in order of occurence in the tangle, if we have encountered a crossing before, we skip it. Prepare an "empty" square matrix $A \in M(n, \{R, G, B, W\})$ where n is the number of crossings, and R, G, B, W represent red, green, blue, and white.

2. For each crossing i in the order, making note of the orientation, we fill entries of A with

   - $A_{ji} = R$
   - $A_{ii} = G$
   - $A_{ki} = B$
   - for all $m \notin \{i, j, k\}$, $A_{mi} = W$,

   where j, k are the crossings left of and right of the crossing i with respect to the over strand, refer to Figure 12 for details

So in other words, each column represents a crossing, and we only consider what is on either side of the crossing. Reading the columns from left to right we get a sense of what crossings are hit. We picked R, G, B, W symbolically, since we wish to plot the matrices, and after a few iterations, any numerical symbols will be illegible.

We plot some incidence matrices of tangles diagrams corresponding to closed knots from the Rolfsen knot table as found in [Rolfsen, 2017]. We see interesting patterns emerge! In Figure 13 we have a few instances of the trefoil tangle diagram with an R1 loop added between some crossings, we see that there is some variation after a number of iterations. This variation does not seem to be consistent since the two lower examples have nearly the same incidence matrices, however the first example differs greatly, e.g. the side bands are closer to the diagonal, and the pattern around the diagonal is different.

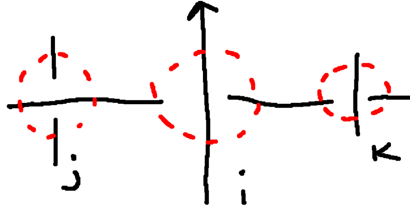


Figure 12: Following the orientation of the crossing i, with respect to the over strand, the left crossing is j and the right crossing is k. The orientation of the crossings j, k is not important.

In Figure 14 we have a few more examples but of different tangles. We see similar kind of behavior that as before, some have similar structures, for example, the trefoil and 4,1 have about the same arrangement of squares, the bands and off diagonal patterns, except the colors change. However if we compare them with 9,10 and 10,23, we see variation in features between those two examples, and the previous trefoil and 4,1.

We see distinct and recognisable structures, yet it is not clear what the mechanism behind their production could be.

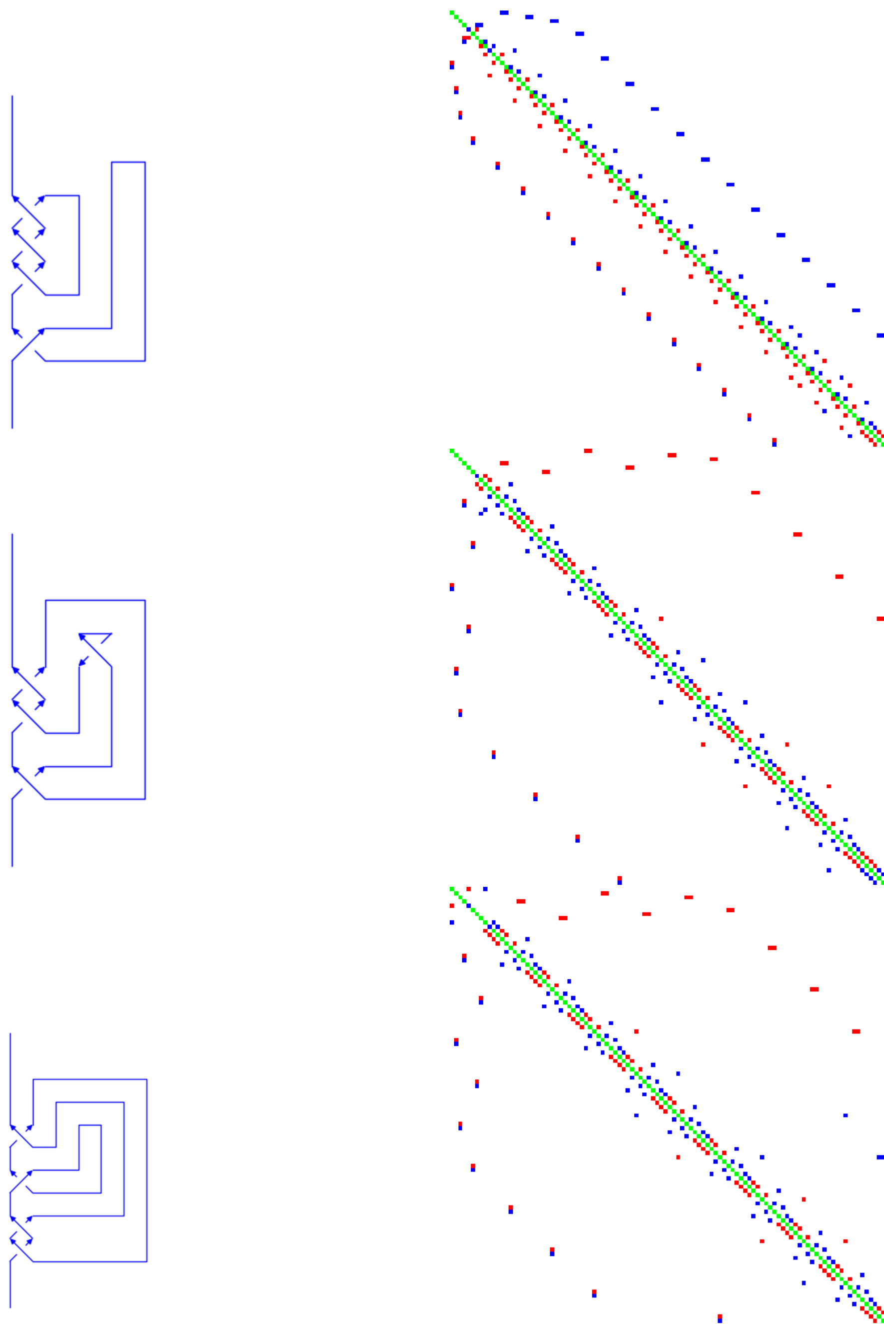
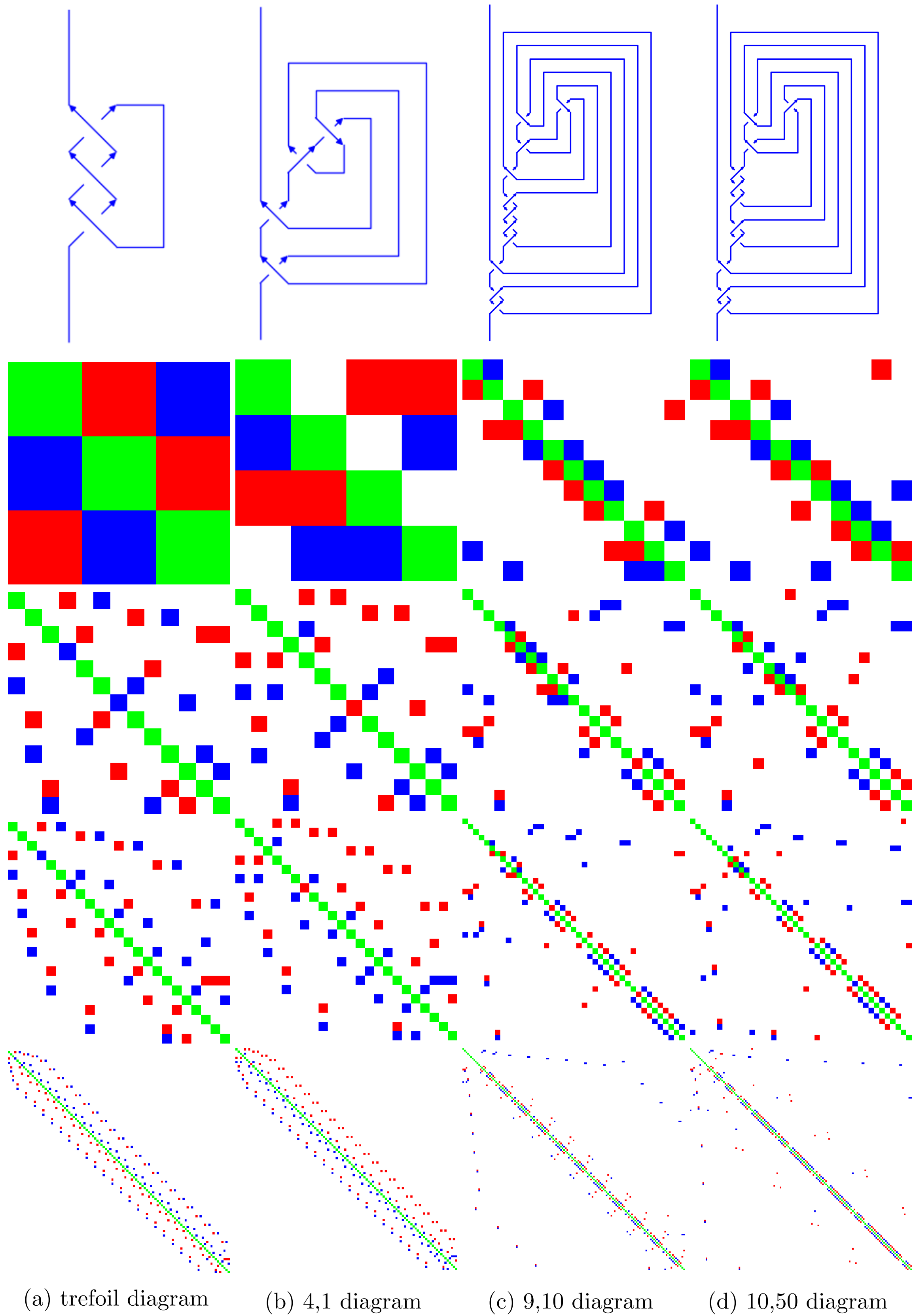


Figure 13: On the left are diagrams of the trefoil with added R1 loops, on the right their respective incidence matrices after 50 iterations of the OU algorithm



(a) trefoil diagram (b) 4,1 diagram (c) 9,10 diagram (d) 10,50 diagram

Figure 14: Comparing incidence matrices of several tangles diagrams under the OU algorithm. Each column is a separate tangle, each column is 0, 5, 10, 50 iterations

6 Conclusions and Further work

6.1 What we did

From Figure 13 and Figure 14 we see that we indeed produce periodic structures with differences from tangle to tangle, and between different starting tangle diagrams. We did not however conclude how these patterns arise.

We only considered a few example diagrams for the trefoil tangle, perhaps for different tangles we will see different results.

Another option is to consider Knots instead of Tangles, so that the glide move may wrap around the starting and end strands.

6.2 Other things to consider

1. Our *OU* algorithm introduced the issue with R1 loops. We chose to ignore the loops in our code, however it can be worthwhile to track the loops, e.g. count their number or position of occurence.

2. The *OU* algorithm gets "stuck" within a cycle, augmenting the algorithm to detect cycles and jump out of them could introduce more structures.

6.3 Convergence

We mentioned a distinction between tame and wild diagrams, yet never suggested any sort of convergence of tame diagrams to wild ones. As the OU algorithm progresses the incidence matrix grows without bound, in such a way producing an infinite structure. If we consider the sequence of incidence matrices as submatrices of infinite matrices, and pick appropriate values for our symbolic R,G,B,W, we may define convergence, and see if it produces any meaningful results.

6.4 Code and accumulated data

For computations and to produce the images (the matrices, and the blue tangle diagrams) we used the `SageMath` package and its respective Knot Theory module. We export the code after the references. We have precomputed and compiled a large body of data, and stored it in a convenient place [Silvans, 2020]. Specifically, videos in `.avi` format of sequences of incidence matrices for tangles corresponding to knots in [Rolfsen, 2017] (we used the Oriented Gauss Code as given by each knot).

References

Adams, Colin. 1994. *Knot Book*. W.H. Freeman.

Bar-Natan, Dror, Dancso, Zsuzsanna, & van der Veen, Roland. 2020. *Over then Under Tangles*.

Lickorish, W.B. Raymond. 1997. *An Introduction to Knot Theory*. Graduate Texts in Mathematics, vol. 175. New York NY: Springer Verlag.

Rolfsen. 2017. *The Rolfsen Knot Table*.

Silvans, Albert. 2020. *Observations on Almost OU Tangles (Repository)*.

In [ ]:

```
%matplotlib inline
```

In [ ]:

```
from itertools import zip_longest, product,groupby
from numpy import pi, e, hstack, vstack, asarray
import numpy as np

from PIL import Image
import matplotlib.pyplot as plt
import os
```

In [ ]:

```
def glide (link):
    '''
    An implementation of the glide move using the oriented
    Gauss code instead of a Dowker-like code.

    Takes as input a Link, outputs a more complicated Link
    '''
    gauss, signs = link.oriented_gauss_code()

    for comp in range(len(gauss)):
        for n in range(len(gauss[comp])-1):
            a, b = gauss[comp][n], gauss[comp][n+1]
            if abs(a) == abs(b) and sign(a) < sign(b):
                pass
            elif sign(a) < sign(b):
                L = len(signs)+1
                R = L+1

                a_sign, b_sign = signs[abs(a)-1], signs[abs(b)-1]

                # add the other crossings first
                # a_idx and b_idx are the locations of the strand with opposite sign
                Acomp_idx = gauss.index(list(filter(lambda x: -a in x, gauss))[0])
                Bcomp_idx = gauss.index(list(filter(lambda x: -b in x, gauss))[0])
                a_idx = gauss[Acomp_idx].index(-a)
                b_idx = gauss[Bcomp_idx].index(-b)

                if a_sign == 1 and b_sign == 1:
                    gauss[Acomp_idx][a_idx] = [L,b,R]
                    gauss[Bcomp_idx][b_idx] = [-R,a,-L]
                    signs += [+1,-1]

                elif a_sign == 1 and b_sign == -1:
                    gauss[Acomp_idx][a_idx] = [L,b,R]
                    gauss[Bcomp_idx][b_idx] = [-L,a,-R]
                    signs += [-1,+1]

                elif a_sign == -1 and b_sign == 1:
                    gauss[Acomp_idx][a_idx] = [R,b,L]
                    gauss[Bcomp_idx][b_idx] = [-R,a,-L]
                    signs += [-1,+1]
                else:
                    gauss[Acomp_idx][a_idx] = [R,b,L]
                    gauss[Bcomp_idx][b_idx] = [-L,a,-R]
                    signs += [+1, -1]

                # swap the orientations of a and b
                signs[abs(a)-1], signs[abs(b)-1] = signs[abs(b)-1], signs[abs(a)-1]
                # swap the signs of a and b
                gauss[comp][gauss[comp].index(a)] *= -1
                gauss[comp][gauss[comp].index(b)] *= -1
                # flatten the
                gauss[Acomp_idx] = flatten(gauss[Acomp_idx])
                gauss[Bcomp_idx] = flatten(gauss[Bcomp_idx])
                break
```

```
    return Link([gauss, signs])
```

In [ ]:

```
def nice_dowker(link):
    '''
    The built-in Dowker notation method does not order the strands,
    or at least I do not see that happening, so I made a version which does that
    '''
    gauss, signs = link.oriented_gauss_code()
    nice_dowker = [ [None,None] for i in range(len(signs))]
    counter = 1
    for i in range(len(gauss)):
        for j in range(len(gauss[i])):
            if sign(gauss[i][j]) == +1:
                nice_dowker[abs(gauss[i][j])-1][1] = counter
            else:
                nice_dowker[abs(gauss[i][j])-1][0] = counter
            counter +=1
    nice_dowker = [ tuple(i) for i in nice_dowker]
    return nice_dowker
```

In [ ]:

```
# some useful function abstractions to make the later code readable
f0 = lambda x: x
f90 = lambda x:  np.rot90(x, k=1,axes=(0,1))
f180 = lambda x:  np.rot90(x, k=2,axes=(0,1))
f270 = lambda x:  np.rot90(x, k=3,axes=(0,1))
rotated = lambda x : [ f(x) for f in [f0, f90, f180, f270]]


def arrange_crossings (link,arrowsize=5):
    '''
    Takes a link, and attempts to organise the crossings on a grid,
    as well as extra other visual elements. Outputs a Graphics object
    that can be plotted or saved

    It works by greedily focusing on the left most entries and depending on
    the structure below we take cases, it looks at each pair of strands in order and:
    (1) tries to connect a crossing to both strands
    (2) if not successful then tries to connect to the left one
    (3) checks if we can close the two strands

    The code is slow and not optimised
    '''

    # we get a "nice" dowker key and then order the crossings
    dowker = nice_dowker(link)
    final = max(flatten(dowker))

    # sort the dowker key by the min strand,
    # also we take into account that we need to sort the signs too
    dowker, signs = zip(*sorted(zip(dowker, link.orientation()), key= lambda x: min(x[0])))
    signs = list(signs)
    # we unwrap the dowker into 2x2 arrays the entries of which
    # are the numbers of the strands. the bottom numbers are coming in,
    # the top are coming out
    crossings = [np.array([[i[0]%final+1,i[1]%final+1],[i[1],i[0]]]) if j==1
                 else np.array([[i[1]%final+1,i[0]%final+1],[i[0],i[1]]])
                 for i,j in zip(dowker, signs)]
    crossings = [(n,i) for n,i in enumerate(crossings)]

    c_rotated = rotated(crossings[0][1])
    accepted = [ n for n, c in enumerate(c_rotated) if c[1][0] == 1 ]
    top_line = flatten([c_rotated[accepted[0]][0,:].tolist(),c_rotated[accepted[0]][1,1]])
    accepted = [ (accepted[0],signs[0],0)]
    crossings.pop(0)
    signs.pop(0)


    cross = [[(0,0), (0,1), (1,1), (1,0)]]
    lines = [[(0,1), (0,1), (0,1)],[(1,1), (1,1), (1,1)],[(1,0), (2,0), (2,1)]]
    extra_bits = []
```

```
    old = len(crossings)
    new = old - 1
    while top_line != [1]:

        checked = 0
        for i, (n, (m,c)) in product(range(len(top_line)-1),enumerate(crossings)):
            c_rotated = rotated(c)
            # check if we can connect two strands
            if top_line[i:i+2] in [ c_rot[1].tolist() for c_rot in c_rotated]:
                checked = 1
                c_rot = c_rotated[[ c_rot[1].tolist() for c_rot in c_rotated].index(top_line[i:i+2])
]
                accepted += [([ c_rot[1].tolist()
                                for c_rot in c_rotated].index(top_line[i:i+2]),signs[n],m)]
                cx, cy = lines[i][2]
                cross += [[(cx,cy),(cx,cy+1),(cx+1,cy+1),(cx+1,cy)]]
                extra_bits += [lines[i],lines[i+1]]
                lines[i] = [(cx,cy+1)]*3
                lines[i+1] = [(cx+1,cy+1)]*3
                lines = [ lin if n==i or n==i+1
                         else lin[:2]+[(lin[2][0],lin[2][1]+1)] for n,lin in enumerate(lines)]
                top_line[i:i+2] = c_rot[0].tolist()
                top_line = list(flatten(top_line))
                crossings.pop(n)
                signs.pop(n)
                break
            # check if we can connect at left strand
            # here we assume that we will never have to check the right most strand in top_line
            elif sum([True if x[1,0] == top_line[i] else False for x in c_rotated]):
                checked = 1
                c_rot = c_rotated[[True if x[1,0] == top_line[i]
                                 else False for x in c_rotated].index(True)]
                accepted += [([True if x[1,0] == top_line[i]
                                 else False for x in c_rotated].index(True),signs[n],m)]
                lines = [ lin[:2]+[(lin[2][0],lin[2][1]+1)] for lin in lines]
                cx,cy = lines[i][2]
                lines = [ [lin[0],(lin[1][0]+2,lin[1][1]),(lin[2][0]+2,lin[2][1]+1)]
                         if (cx,cy) not in lin and lin[2][0] > cx
                         else lin for lin in lines]
                lines = [ [lin[0],(lin[1][0],lin[1][1]),(lin[2][0],lin[2][1]+1)]
                         if (cx,cy) not in lin and lin[2][0] < cx
                         else lin for lin in lines]
                cross += [[(cx,cy),(cx,cy+1),(cx+1,cy+1),(cx+1,cy)]]
                extra_bits += [lines[i]]
                lines = lines[:i] +[[(cx,cy+1)]*3,[(cx+1,cy+1)]*3, [(cx+1,cy),(cx+2,cy),(cx+2,cy+1)]
] + lines[i+1:]
                top_line[i] = c_rot[0].tolist() + [c_rot[1,1]]
                top_line = list(flatten(top_line))
                crossings.pop(n)
                signs.pop(n)
                break
            elif top_line[i] == top_line[i+1]:
                checked=1
                # add the 'caps', i.e. connect the strands on top
                extra_bits += [lines[i]+list(reversed(lines[i+1]))]
                top_line.pop(i+1)
                top_line.pop(i)
                lines.pop(i+1)
                lines.pop(i)
                lines = [ lin[:2]+[(lin[2][0],lin[2][1]+1)] for lin in lines]
                break

        if crossings == []:
            for i in range(len(top_line)-1):
                if top_line[i] == top_line[i+1]:
                    checked = 1
                    # add the 'caps', i.e. connect the strands on top
                    extra_bits += [lines[i]+list(reversed(lines[i+1]))]
                    top_line.pop(i+1)
                    top_line.pop(i)
                    lines.pop(i+1)
                    lines.pop(i)
                    lines = [ lin[:2]+[(lin[2][0],lin[2][1]+1)] for lin in lines]
                    break

        elif checked == 0:
            for n, (m,c) in enumerate(crossings):
```

```
                c_rotated = rotated(c)
                if sum([True if x[1,0] == top_line[-1] else False for x in c_rotated]):
                    c_rot = c_rotated[[True if x[1,0] == top_line[-1]
                                       else False for x in c_rotated].index(True)]
                    accepted += [([True if x[1,0] == top_line[-1]
                                       else False for x in c_rotated].index(True),signs[n],m)
                    lines = [ lin[:2]+[(lin[2][0],lin[2][1]+1)] for lin in lines]
                    cx,cy = lines[-1][2]
                    lines = [ [lin[0],(lin[1][0]+2,lin[1][1]),(lin[2][0]+2,lin[2][1]+1)]
                             if (cx,cy) not in lin and lin[2][0] > cx
                             else lin for lin in lines]
                    lines = [ [lin[0],(lin[1][0],lin[1][1]),(lin[2][0],lin[2][1]+1)]
                             if (cx,cy) not in lin and lin[2][0] < cx
                             else lin for lin in lines]
                    cross += [[(cx,cy),(cx,cy+1),(cx+1,cy+1),(cx+1,cy)]]
                    extra_bits += [lines[-1]]
                    lines = lines[:-1] +[[(cx,cy+1)]*3,[(cx+1,cy+1)]*3, [(cx+1,cy),(cx+2,cy),(cx+2,c
y+1)]]
                    top_line[-1] = c_rot[0].tolist() + [c_rot[1,1]]
                    top_line = list(flatten(top_line))
                    crossings.pop(n)
                    signs.pop(n)
                    break

    G=Graphics()

    for n,(d,s,m) in enumerate(accepted):
        if d==0 and s==1:
            G += arrow2d(cross[n][3],cross[n][1],width=1,arrowsize=arrowsize,zorder=1)
            G += circle(((cross[n][2][0]+cross[n][0][0])/2,(cross[n][2][1]+cross[n][0][1])/2),0.25,fi
ll=True, facecolor='white',edgecolor='white',zorder=1)
            G += arrow2d(cross[n][0],cross[n][2],width=1,arrowsize=arrowsize,zorder=2)
        if d==0 and s==-1:
            G += arrow2d(cross[n][0],cross[n][2],width=1,arrowsize=arrowsize,zorder=1)
            G += circle(((cross[n][2][0]+cross[n][0][0])/2,(cross[n][2][1]+cross[n][0][1])/2),0.25,fi
ll=True, facecolor='white',edgecolor='white',zorder=1)
            G += arrow2d(cross[n][3],cross[n][1],width=1,arrowsize=arrowsize,zorder=2)
        if d==1 and s==1:
            G += arrow2d(cross[n][2],cross[n][0],width=1,arrowsize=arrowsize,zorder=1)
            G += circle(((cross[n][2][0]+cross[n][0][0])/2,(cross[n][2][1]+cross[n][0][1])/2),0.25,fi
ll=True, facecolor='white',edgecolor='white',zorder=1)
            G += arrow2d(cross[n][3],cross[n][1],width=1,arrowsize=arrowsize,zorder=2)
        if d==1 and s==-1:
            G += arrow2d(cross[n][3],cross[n][1],width=1,arrowsize=arrowsize,zorder=1)
            G += circle(((cross[n][2][0]+cross[n][0][0])/2,(cross[n][2][1]+cross[n][0][1])/2),0.25,fi
ll=True, facecolor='white',edgecolor='white',zorder=1)
            G += arrow2d(cross[n][2],cross[n][0],width=1,arrowsize=arrowsize,zorder=2)
        if d==2 and s==1:
            G += arrow2d(cross[n][1],cross[n][3],width=1,arrowsize=arrowsize,zorder=1)
            G += circle(((cross[n][2][0]+cross[n][0][0])/2,(cross[n][2][1]+cross[n][0][1])/2),0.25,fi
ll=True, facecolor='white',edgecolor='white',zorder=1)
            G += arrow2d(cross[n][2],cross[n][0],width=1,arrowsize=arrowsize,zorder=2)
        if d==2 and s==-1:
            G += arrow2d(cross[n][2],cross[n][0],width=1,arrowsize=arrowsize,zorder=1)
            G += circle(((cross[n][2][0]+cross[n][0][0])/2,(cross[n][2][1]+cross[n][0][1])/2),0.25,fi
ll=True, facecolor='white',edgecolor='white',zorder=1)
            G += arrow2d(cross[n][1],cross[n][3],width=1,arrowsize=arrowsize,zorder=2)
        if d==3 and s==1:
            G += arrow2d(cross[n][0],cross[n][2],width=1,arrowsize=arrowsize,zorder=1)
            G += circle(((cross[n][2][0]+cross[n][0][0])/2,(cross[n][2][1]+cross[n][0][1])/2),0.25,fi
ll=True, facecolor='white',edgecolor='white',zorder=1)
            G += arrow2d(cross[n][1],cross[n][3],width=1,arrowsize=arrowsize,zorder=2)
        if d==3 and s==-1:
            G += arrow2d(cross[n][1],cross[n][3],width=1,arrowsize=arrowsize,zorder=1)
            G += circle(((cross[n][2][0]+cross[n][0][0])/2,(cross[n][2][1]+cross[n][0][1])/2),0.25,fi
ll=True, facecolor='white',edgecolor='white',zorder=1)
            G += arrow2d(cross[n][0],cross[n][2],width=1,arrowsize=arrowsize,zorder=2)


    # add a little length to the outgoing strand
    lines = [ lin[:2]+[(lin[2][0],lin[2][1]+1)] for lin in lines]
    for lin in lines+extra_bits:
        G += line(lin)

    # add a little length to the incoming strand
    G += line([(0,0),(0,-2)])
```

```
    G.set_aspect_ratio(1)
    G.axes(False)

    return G
```

In [ ]:

```
knot = Knots().from_table(3,1)

for i in range(10):
    knot = glide(knot)
# drawing the arrows is an issue because it's not clear how to scale them consistently,
# so we coded the function in a way one can specify it on the scaling on their own.
# we notice a general rule of thumb that works is to specify scaling as 1/number of crossings
G = arrange_crossings(knot,arrowsize=1/len(knot.orientation()))

G.show(figsize=[5,5])
```

In [ ]:

```
#for 1 tangles
T.<t> = LaurentPolynomialRing(ZZ)

coloring = {0:[255,255,255], # white
            T(t):[255,0,0],  # red
            T(-1):[0,0,255], # blue
            T(1-t):[0,255,0]}# green

colorize_incd_mat = lambda m: [ [ coloring[j] for j in i] for i in m]

def get_incd_mat(link):

    gauss, signs = link.oriented_gauss_code()
    gauss = gauss[0]
    glen = len(gauss)

    columns = []
    for i in gauss:
        if abs(i) not in columns:
            columns += [abs(i)]
    # we pad the gauss code with itself, so that it's easier to find the over strands later
    pgauss = [gauss[-1]] + gauss + [gauss[0]]

    entries = [ [abs(pgauss[ent])-1,
                 abs(pgauss[ent+2])-1]
               for ent in [ gauss.index(-i) for i in columns]]

    size = len(signs)
    mat = np.full((size,)*2, T(0))

    for c,[i,j] in zip([ i-1 for i in columns],entries):
        s = signs[c]
        mat[c][c] = T(1-t)
        if s == 1:
            mat[j][c] = T(t)
            mat[i][c] = T(-1)
        else:
            mat[j][c] = T(-1)
            mat[i][c] = T(t)

    return mat
```

In [ ]:

```
# code to produce incidence matrices of the tangle diagrams
import os

knot_names = [ (n,j) for n,i in enumerate([1,1,2,3,7,21,49,165],3) for j in range(1,i+1)]
pas = 0
for knot in knot_names:
    frames = 50
    video_name = "video_"+str(knot[0])+"_"+str(knot[1])+"_"+str(frames)+".avi"
    knot = Knots().from_table(*knot).mirror_image()
```

```
    for i in range(frames):
        nodes = arrange_crossings(knot)

        img = Image.fromarray(np.array(colorize_incd_mat(get_incd_mat(knot)),dtype='uint8'),'RGB')
        img = img.resize((600,600), resample=Image.NEAREST)#Image.BOX)
        image_name = './' + str(i) + '.png'
        img.save(image_name)
        knot = glide(knot)

    # need to have ffmeg installed on system
    os.system("ffmpeg -f image2 -r 2 -i ./%01d.png -vcodec huffyuv -y ./"+video_name)
    os.system("rm ./*.png")
```

In [ ]:

```
# checking how the algorithm differs for the same tangle but with an added R1 move in between

knot = Knots().from_table(3,1)
knot, signs = knot.oriented_gauss_code()
knot = knot[0]
c = len(signs)


knot_names = [ [Knot([[knot[:i]+[c+1,-(c+1)]+knot[i:]],signs + [1]]),
                Knot([[knot[:i]+[c+1,-(c+1)]+knot[i:]],signs + [-1]])] for i in range(len(knot)+1)]

knot_names = flatten(knot_names)

for n, knot in enumerate(knot_names):

    frames = 50
    G = arrange_crossings(knot)
    G.save('3_1_ed_'+str(n)+'.png')

    for i in range(frames):
        knot = glide(knot)

    img = Image.fromarray(np.array(colorize_incd_mat(get_incd_mat(knot)),dtype='uint8'),'RGB')
    img = img.resize((600,600), resample=Image.NEAREST)#Image.BOX)
    image_name = './tests_with_3_1/' + str(n) +'_'+ str(frames) + '.png'
    img.save(image_name)
```