Implementation of the Kauffman Polynomial in SageMath

Antonio De Lucreziis

Abstract. In this project we write implement from scratch the Kauffman polynomial in Python. First we introduce problems in computational knot theory and describe various representations of knots and links and find a good representation to use for the algorithm. We then describe in-depth the algorithm for computing the Kauffman polynomial and how to implement it in Python. Finally we try the algorithm on various knots and links and compare the results with the ones from the KnotInfo Database.

Contents

1. Introduction

Actually we don't like Python so we will be using Rust and then write bindings for Python that can be used in SageMath.

1.1. The Kauffman Polynomial

The Kauffman polynomial L is a two-variable polynomial invariant of unoriented knots and links in 3-dimensional space. It is defined using *Skein relations*, more precisely an implicit functional equation.

The defining axioms of the Kauffman polynomial are the following, given a link diagram K we have $L_{K(a,z)} \in \mathbb{Z}[a, a^{-1}, z, z^{-1}]$ and:

i) If K and K' are two equivalent up to regular isotopy, then $L_K(a, z) = L_{K'}(a, z)$.

ii) We have the following identities:

- $L\Big(\times\Big) + L\Big(\times\Big) = z\Big(L\Big(\smile\,\frown\Big) + L\Big(\,\Big)\;\Big(\,\Big)\Big)$

- $L\Big($  $\Big) = 1$
- $L\Big($  $\Big) = aL\Big($  $\Big)$
- $L\Big($  $\Big) = a^{-1}L\Big($  $\Big)$

We will later be seeing that the Kauffman polynomial can be defined in a more explicit way, using a recursive definition that is the one we will be using to derive our algorithm.

1.2. Computational Knot Theory

The first problem in computational knot theory is to find a good representation for knots and links. There are various common representations in the literature, such as:

- **Gauss codes**:

  This is very simple to generate, we just need to label each crossing with a number and then write down the sequence of numbers in the order they appear by walking along the knot. This has the problem that it is not unique, for example it does not distinguish between the trefoil knot and its mirror image.

- **Signed Gauss codes**:

  This is an improvement over the previous representation, on the second occurrence of a number we use a $+$ or $-$ sign to indicate the handedness of the crossing.

- **Planar diagram codes (PD codes)**, this is the one we will be using in this project:

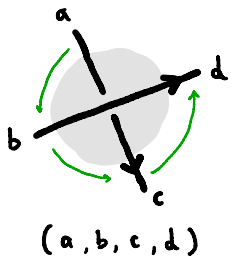
  The PD code for a link is generated by labelling each arc of the link with a number. Then we choose a starting point and walk along the link, when we pass at an over-crossing for the current strand we write down a 4-uple of counter-clockwise numbers for the arc incident to the crossing.

There are also other codes like **Braid representations** and **DT (Dowker-Thistlethwaite) Codes** we will not be using in this project.

1.2.1. PD CODES

The main source for the section is the article on PD notation from KnotInfo. The PD code for a link is generated by labelling each arc of the link with a number. Then we choose a starting point for each component and process each component in order.

For each component we walk along it from the starting point in the component direction. When we pass at a crossing that is an over-crossing for the current strand we write down a 4-uple of counter-clockwise numbers for the arc incident to the crossing starting from the *entering under-crossing* arc.

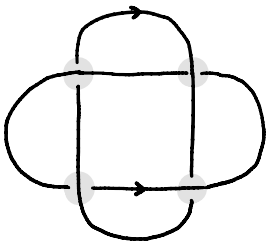


Algorithm:

```
Input: An oriented link diagram with starting points on each component
Output: List<(Nat, Nat, Nat, Nat)>

- Choose an ordering for the components and starting point for each component
- Label each arc with a number
- For each component:
  - Walk along it from the starting point in its orientation
  - At each crossing, when at an over-crossing for the current strand
    - Write down a 4-uple of counter-clockwise numbers for the arc incident
      to the crossing starting from the entering under-crossing arc
```

Let's see an example of how to construct the PD code for the following link diagram



First let's choose a starting point for each component and label accordingly the arcs of the link. We will use the following convention:

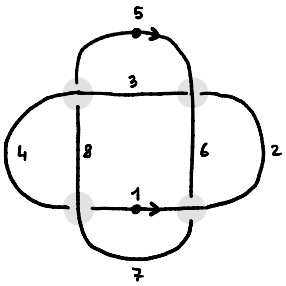
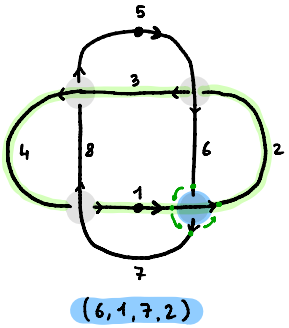
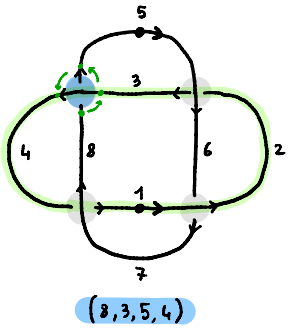


FIGURE 3. Oriented link with starting points and edge labels.

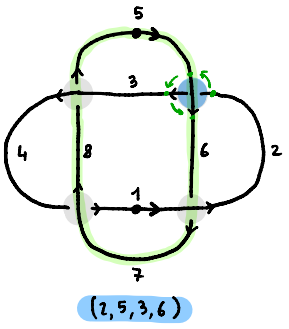
Now we can start processing each component of the link by walking along it in its orientation and writing down the over-crossings we encounter.



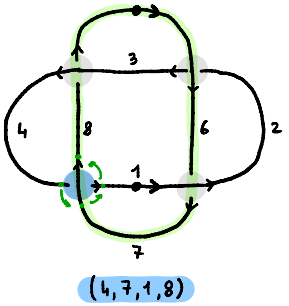
First link component
First over-crossing
⇒ `(6,1,7,2)`



First link component
Second over-crossing
⇒ `(8,3,5,4)`



Second link component
First over-crossing
⇒ `(2,5,3,6)`



Second link component
Second over-crossing
⇒ `(4,7,1,8)`

Every directed crossing appears only once as an over-crossing so this algorithm terminates when all crossings have been visited. So the final PD code for this link is

```
[(6,1,7,2), (8,3,5,4), (2,5,3,6), (4,7,1,8)]
```

1.2.2. SG Codes

Gauss originally developed a notation called **Gauss codes** based on labelling each crossing of a knot with a number and keeping track of when we walk an over-crossing or an under-crossing using a sign. This produces a list of numbers where each number appears exactly twice with different signs. This has a few problems like the fact that this doesn't distinguish a knot vs its mirror.

This is solved by **Signed Gauss Codes**[1] where we also store the information about the handedness of a crossing.

More precisely this is constructed by the following: for each component we walk along it, when we pass at a crossing we write a tuple $(\pm i, \pm 1)$ where the first component is the index of the crossing with a sign indicating if this is an over-crossing or under-crossing, the second sign (that can be added in a second pass over the loop) is given by the handedness of the crossing using the following convention

$$\varepsilon\left(\diagup\!\!\!\diagdown\right) = +1 \qquad \varepsilon\left(\diagdown\!\!\!\diagup\right) = -1$$

Algorithm:

```
Input: An oriented link diagram with starting points on each component
Output: List<List<(Int, Int)>>

- Label each crossing with a number in order
- For each component:
  - Walk along it from the starting point in its orientation
  - At each crossing, write a tuple with components
    - +i or -i if this is an over-crossing or under-crossing
    - +1 or -1 if this is a left-handed or right-handed
```

Converting one code to the other is not too much work as one just need to first do a labelling step to convert crossing labels and then convert the over/under-strand and left/right-handedness relations between the two notations.

1.2.3. Comparison of PD and SG codes

We will now see how SG codes are better suited for the manipulations (switching and splicing) we need to do on a link. Indeed the KnotTheory Mathematica package when computing the Kauffman polynomial converts the input link from **PD code** into **SG code** as this is better suited for doing manipulations directly on the crossings.

- **Switch**
  - **SG codes** – This is just two *sign swaps* on each of the two occurrences of the over and under strand.
  - **PD codes** – This is more involved and requires *cycling* the crossing from (i, j, k, l) to one of (l, i, j, k) or (j, k, l, i) based on the crossing sign and *relabelling* the whole affected components as PD codes heavily rely on the sequentiality of the indices to tell the direction and end of a component.
- **Splicing**
  - **SG codes** – This is just a matter of splitting, reversing and rejoining lists correctly, this is a bit involved and will be covered more in depth later.
  - **PD codes** – This can be approached in various ways more or less performant.

[1] Gauss Notation on Wikipedia

One can add a meaning to pairs (i, j) symbols to the original sequence of 4-uples called "path" elements (the KnotTheory package has this extension of the PD notation with the `P[i, j]` element) that tell we joined the arc i with the arc j. This gives an heterogeneous list of elements that is more complex to handle efficiently in classical programming languages different from Mathematica.

Another approach to keep list homogeneous is to manually splice the crossing and do a relabelling of all the arcs at each step. This causes in the worst case a continuos relabelling of all the crossings in the link at every splice and is not very efficient as our algorithm needs to be able to do splicing as fast as possible.

Another downside of PD codes is the ordering of the crossings in-memory, walking along a component might require various jumps along the list. SG codes on the other hand have already each component in the correct order and can be walked linearly without jumps.

Finally SG codes are also more "space efficient". Let N be a number of bits to encode a natural number of maximum fixed size (e.g. $N = 8$ for using `uint8` to encode numbers), for a link with n crossings and k components

- PD codes use $\approx 4n \times N$ bits of information, four numbers for each item.
- SG codes use $\approx 2n \times (N + 2) + k \times \lceil \log_2(n) \rceil$ bits of information. Each crossing appears twice and we store its id and over/under and handedness information, we also need to store the structure of the list with $k \times \lceil \log_2(n) \rceil$ more bits.

So PD codes are simpler and compact to store (and generate from a diagram) but SG codes are more space efficient and easy to manipulate.

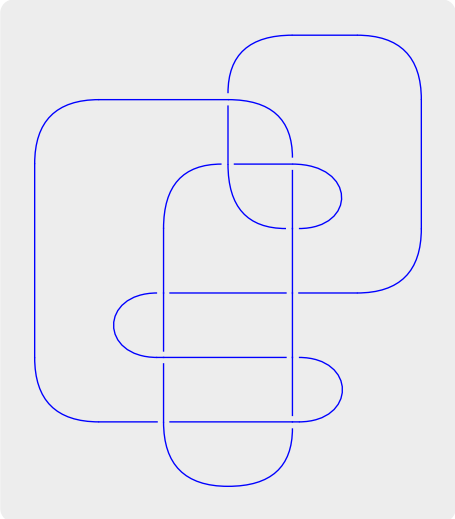
1.2.4. Link reconstruction from code

We briefly mention that reconstructing a link from a PD or SG code is not trivial and there are various approaches used by various softwares that can be used for this task.

◆ Linear Integer Programming

For example the KnotTheory package in Sage has a `Link.plot()` method that thats a link that can be constructed using a PD code and then plots it as follows

```
# The "monster" unknot
L = Link([[ 3,  1,  2,  4], [ 8,  9,  1,  7], [ 5,  6,  7,  3], [ 4, 18,  6,  5],
          [17, 19,  8, 18], [ 9, 10, 11, 14], [10, 12, 13, 11], [12, 19, 15, 13],
          [20, 16, 14, 15], [16, 20, 17,  2]])
L.plot()
```



Sage internally uses a *mixed integer linear programming* (MILP) solver to generate a knot diagram from a PD code. Another library called Spherogram instead uses *network flows*. The problem here is to find an orthogonal presentation for the link with the *minimum number of left and right bends*[2].

◆ Planar Graph Embeddings

Another approach used by KnotFolio is based on Tutte embeddings. A **Tutte embedding or barycentric embedding** of a simple, 3-vertex-connected, planar graph is a crossing-free straight-line embedding with the properties that the outer face is a convex polygon and that each interior vertex is at the average (or barycenter) of its neighbors' positions.

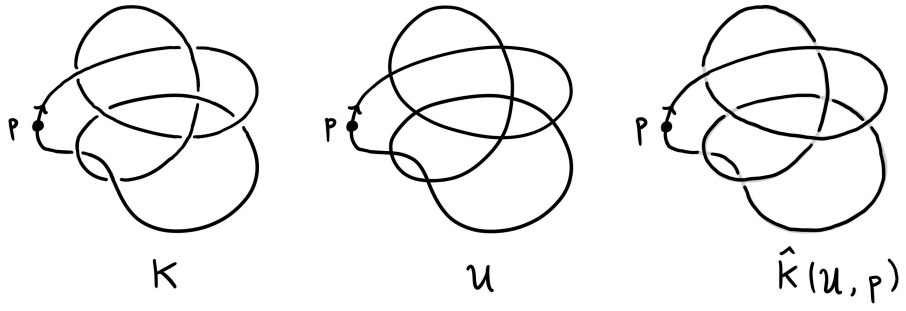
This condition that every point is the average of its neighbors can be easily expressed as a system of linear equations where some points on a chosen outer face have been fixed. When the graph is planar and 3-vertex-connected the linear system is non degenerate and has a unique solution.

[2] A better heuristic for area-compaction of orthogonal representations

1.3. **Algorithm for computing the Kauffman Polynomial**

Let's now recap the main formal algorithm for computing the Kauffman polynomial.

Definition 1. Let K be an un-oriented link. We denote by $\hat{K}(p)$ the **standard unknot** for K where p is a *directed starting point*. This is built by considering the planar shadow U of K and walking along U starting from p and making each crossing an over-crossing when passing on it the first time.



Definition 2. Let's give a name to the following knot modifications for K near a specific crossing i

K	S_iK	E_iK	e_iK
original	*switch*	*h-splice*	*v-splice*

Let now K be an oriented link with n components so $K = K_1 \cup \cdots \cup K_n$.

Definition 3. Let K abd $\lambda = (\lambda_n, ..., \lambda_0)$ a sequence of indices of crossing of K and let i be an index of one of the crossings, let's define the following actions

- $A_i^\lambda K \coloneqq E_i S_{\lambda_i} \cdots S_{\lambda_0} K$
- $B_i^\lambda K \coloneqq e_i S_{\lambda_i} \cdots S_{\lambda_0} K$
- Then let λ be a sequence of indices that bring K to $\hat{K}$ so that $\hat{K}(\lambda) \coloneqq S_{\lambda_n} \cdots S_{\lambda_0} K$ and define

$$\sum_K(\lambda) = \sum_{i=0}^{n} (-1)^i \big(L\big(A_i^\lambda K\big) + L\big(B_i^\lambda K\big)\big)$$

$$\Omega_K(\lambda) = (-1)^{|\lambda|+1} L_{\widehat{K}(\lambda)} + z \sum_K(\lambda)$$

Algorithm. Here follows the algorithm that computes $L_K(a, z)$.

i) If $K = \hat{K}$ is a *standard unknot* then $L_K(a, z) \coloneqq a^{w(K)}$

ii) If K_1 *overlies* K_2, let $d \coloneqq (a + a^{-1})/z - 1$ and then

$$L(K_1 \cup K_2) \coloneqq dL(K_1)L(K_2)$$

iii) If $K = K_1 \cup \cdots \cup K_n$

- If a K_i *overlies* another another component than apply (ii).

- If no K_i *overlies* all others let $p_1, ..., p_n$ be *directed starting points* on $K_1, ..., K_n$ and let $\overline{p}_i$ be the same *directed starting point* with the opposite direction of p_i on K_i. Let $\lambda(p_i)$ the sequence of under-crossings of K_i with $K - K_i$ so that $\hat{K}(\lambda(p_i)) = K_i \sqcup (K - K_i)$ so that K_i *overlies* the rest of the components. At this point we can define L_K as

$$L_K(a, z) := \frac{1}{2n}\left[\sum_{i=1}^{n}\sum_{q=p_i,\overline{p}_i}\left((-1)^{|\lambda(q)|+1}dL_{K_i}L_{K-K_i} + z\sum_{K}(\lambda(q))\right)\right]$$

- If K is a single component then let p be a directed starting point on K and $\overline{p}$ the one with opposite direction. Let $\lambda(p)$ the switching sequence that brings it to the standard unknot $\hat{K}$ and define

$$L_K(a, z) := \frac{1}{2}\left[\sum_{q=p,\overline{p}}\left((-1)^{|\lambda(q)|+1}L\Big(\hat{K}(\lambda(q))\Big) + z\sum_{K}(\lambda(q))\right)\right]$$

1.4. Python Implementation

The approach has been a mix of bottom-up and top-down. First we defined a couple of classed `SignedGaussCode` and `PDCode` to work with these codes and easily convert between each other.

1.4.1. SG Codes

We are now going to walk thorough the class that lets use work nicely with **SG codes**. The the classes we are going to use are all *frozen data-classes* to ensure immutability and enforce a more functional programming style.

```python
@dataclass(frozen=True)
class SignedGaussCodeCrossing:
    id: int
    over_under: typing.Literal[+1, -1]
    handedness: typing.Literal[+1, -1]

    def is_over(self) -> bool: ...
    def is_under(self) -> bool: ...
    def is_left(self) -> bool: ...
    def is_right(self) -> bool: ...
    def opposite(self) -> SignedGaussCodeCrossing: ...
```

```python
@dataclass(frozen=True)
class SignedGaussCode:
    components: list[list[SignedGaussCodeCrossing]]

    def writhe(self): ...
    def reverse(self): ...
    def mirror(self): ...
    def to_std_unknot(self) -> SignedGaussCode: ...
    def std_unknot_switching_sequence(self) -> list[int]: ...
    def apply_switching_sequence(self, seq: list[int]) -> SignedGaussCode: ...
    def splice_h(self, id: int): ...
    def splice_v(self, id: int): ...
    def switch_crossing(self, id: int): ...
```

◆ **Writhe**

One of the first important things we need is to compute the **writhe** $w(K)$ of a link, this can easily be done with signed gauss codes as its a list of tuples where the second entry is the crossing sign. Let L be an oriented link with components $C_1, ..., C_k$ each with crossings $c_{i,j}$ with $i = 1, ..., k$ and $j = 1, ..., |C_i|$.

Let's notice that here each crossing appears twice, once as over-crossing and once as an under-crossing this is the reason for the $1/2$ in the following formula. By $\varepsilon(c)$ we refer to the sign (or handedness) of the crossing at c.

$$w(L) = \frac{1}{2} \sum_{c \text{ crossing}} \varepsilon(c) \quad \rightsquigarrow$$

```python
def writhe(self):
    return sum(
        c.handedness # ⇒ +1 or -1
        for component in self.components
        for c in component
    ) // 2
```

◆ Standard Unknot

The next building block for computing the Kauffman polynomial is detecting and computing the **standard unknot or unlink**. Formally this is done by taking the *planar shadow* and a directed starting point on it. Then we can walk along the shadow and make each crossing an over-crossing when passing on it on the first time.

On the other hand our algorithm directly works with switching sequences λ that bring a knot K to its standard unknot $\hat{K}$. We wrote methods to directly compute and apply these switching sequences.

```python
def std_unknot_switching_sequence(self) -> list[int]:
    visited_crossings: set[int] = set()
    switched_crossings: list[int] = []

    for component in self.components:
        for crossing in component:
            if crossing.id not in visited_crossings:
                if crossing.is_under():
                    switched_crossings.append(crossing.id)
                visited_crossings.add(crossing.id)

    return switched_crossings
```

The `std_unknot_switching_sequence` method just walks along each component in its orientation marking what switches have to be made to bring that link to its standard unknot.

```python
def apply_switching_sequence(self, seq: list[int]) -> SignedGaussCode:
    return SignedGaussCode([
        [
            crossing.opposite(invert_handedness=True)
            if crossing.id in seq else crossing
            for crossing in component
        ]
        for component in self.components
    ])
```

Applying a switching sequence is just a matter of flipping the crossings that are in the sequence. This is also how the `switch_crossing(id: int)` method works.

◆ Crossing Splices

The splicing code is more involved due to the number of cases to analyze, let's first see formally what we need to do.

We have all the following cases, first we can assume the *entering over strand* is in the top left corner of a diagram (this can be done by applying locally a small local isotopy). Then we have the following cases

- Splice type: horizontal or vertical
- Crossing sign: left-handed or right-handed
- Crossing type: self-crossing or crossing between two different strands

So we have a total of $2 \times 2 \times 2 = 8$ cases to analyze. The following diagram shows all the possible cases for horizontal splicing for *signed gauss codes*.

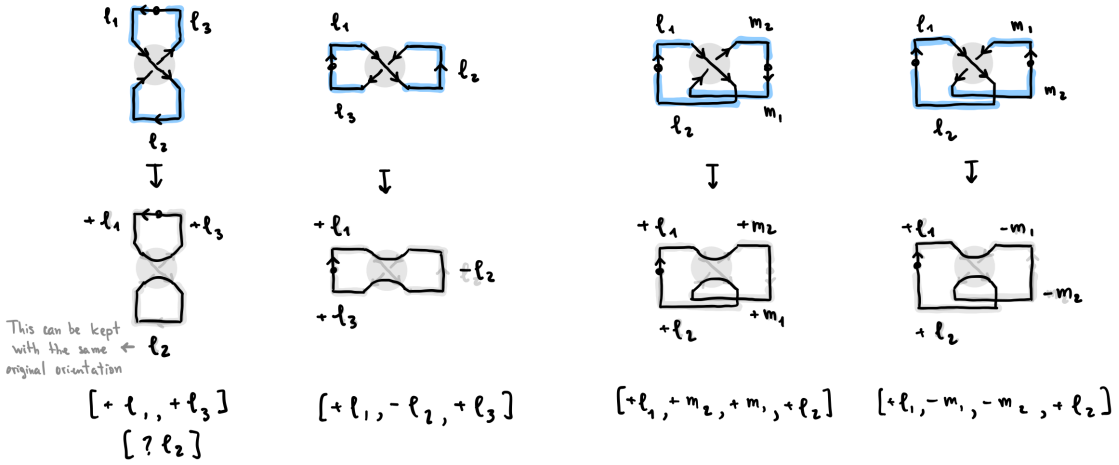


FIGURE 5. Cases for horizontal splicing

Let's explain this diagram a bit, each label is a part of the list for the component, the "−" sign tells is that part is walked in opposite order and must reversed in the final list.

The first two cases in the top left are the ones where the splice happens on a self-crossing that is the crossing is with two parts of the same strand. We can assume the starting point is before the over-strand, the result is the same as we can just rotate the list to get in this configuration. So if this component has n crossings and i and j are the indices of the over-strand crossing and the under-strand crossing respectively the code will be

$$[\ \cdots, [\ C_1,\ \cdots, C_i,\ \cdots, C_j,\ \cdots, C_{2n}\ ],\ \cdots\ ]$$

where $C_k = (c_k, s_k)$ as a pair of crossing **id** and **sign**. To apply the splice we remove the crossings (c_i, s_i) and (c_j, s_j) from that component list and rejoin following the orientation of the strand starting from the starting point.

- In the **positive crossing** case the horizontal splice splits the component in two, the first one composed of the first and third part one after the other and another one composed only of the second part. More precisely we have the following two new components

$$[\ C_1,\ \cdots, C_{i-1},\ C_{j+1},\ \cdots, C_{2n}\ ]$$
$$[\ C_{i+1},\ \cdots, C_{j-1}\ ]$$

- In the **negative crossing** case we first walk on the first part of the list, then we walk the second part in reverse and finally the third part. So the new component will be

$$[\, C_1, \cdots, C_{i-1}, \underbrace{C_{j-1}, C_{j-2}, \cdots, C_{i+2}, C_{i+1}}_{\text{reversed part}}, C_{j+1}, \cdots, C_{2n} \,]$$

Handling the reversed part is actually more involved than just reversing the list of crossings. We also need to correct all the signs of the crossings to account for the new orientation. To do this we need to flip the crossing sign of all crossing ids that occur in this part we are reversing. Notice this can end up even alter signs of crossings in other components so we need to be careful. The code that handles with reversing is the following

TODO: Code snippet

The code for the **vertical splices** is omitted as the cases are the same as for the horizontal splices with a minor change. All the cases for vertical splices are shown below in the following diagram and we can see that the output lists are the same as in the previous splice case just switched based on the crossing sign.

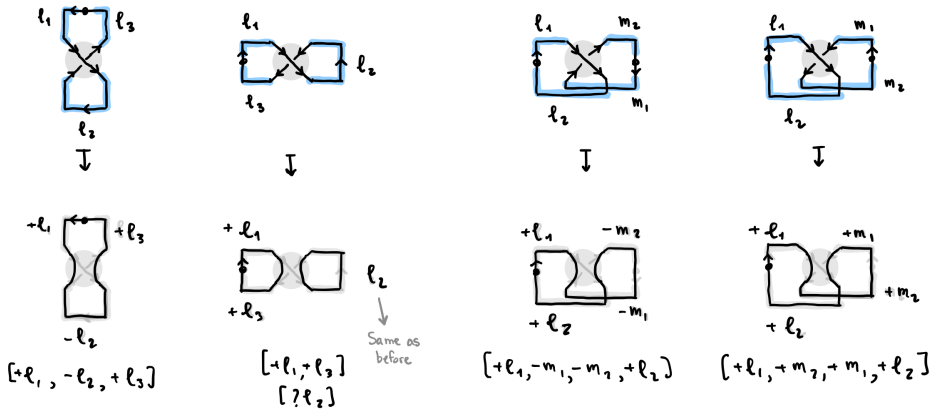


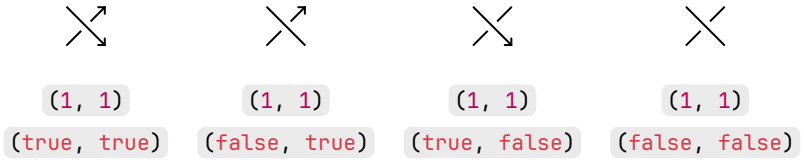
FIGURE 6. Cases for vertical splicing

So the final code is just a conversion of all this cases to list slicing and re-joining with the appropriate crossings removed and signs updated correctly.

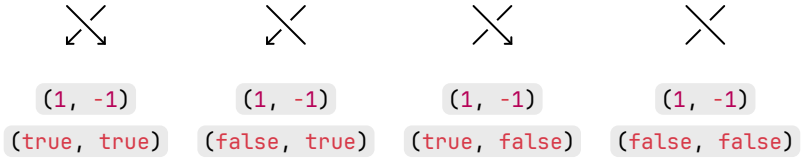
2. Appendix

All combinations of `skein-generic` typst function

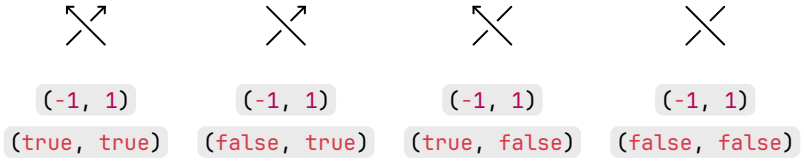
Kind: `"over"`



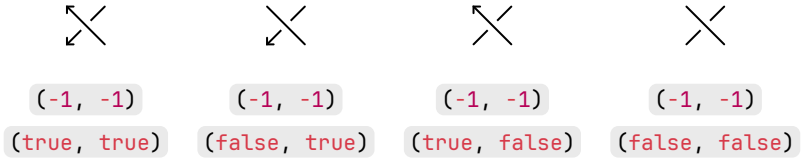
`(1, 1)` `(true, true)` | `(1, 1)` `(false, true)` | `(1, 1)` `(true, false)` | `(1, 1)` `(false, false)`



`(1, -1)` `(true, true)` | `(1, -1)` `(false, true)` | `(1, -1)` `(true, false)` | `(1, -1)` `(false, false)`

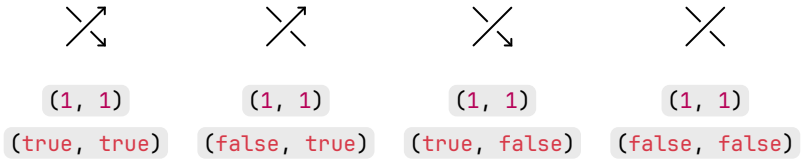


`(-1, 1)` `(true, true)` | `(-1, 1)` `(false, true)` | `(-1, 1)` `(true, false)` | `(-1, 1)` `(false, false)`

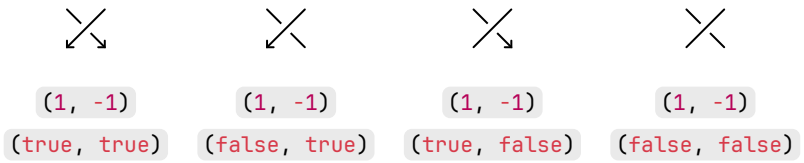


`(-1, -1)` `(true, true)` | `(-1, -1)` `(false, true)` | `(-1, -1)` `(true, false)` | `(-1, -1)` `(false, false)`

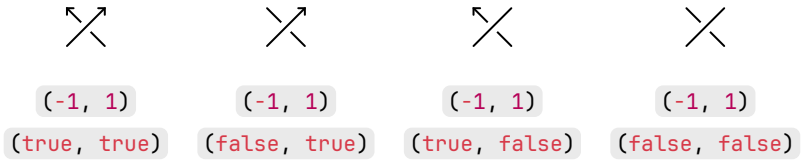
Kind: `"under"`



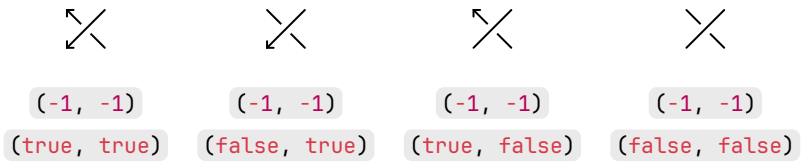
`(1, 1)` `(true, true)` | `(1, 1)` `(false, true)` | `(1, 1)` `(true, false)` | `(1, 1)` `(false, false)`



`(1, -1)` `(true, true)` | `(1, -1)` `(false, true)` | `(1, -1)` `(true, false)` | `(1, -1)` `(false, false)`



`(-1, 1)` `(true, true)` | `(-1, 1)` `(false, true)` | `(-1, 1)` `(true, false)` | `(-1, 1)` `(false, false)`



`(-1, -1)` `(true, true)` | `(-1, -1)` `(false, true)` | `(-1, -1)` `(true, false)` | `(-1, -1)` `(false, false)`