Implementation of the Kauffman Polynomial in Python

Antonio De Lucreziis

Abstract. In this project we write implement from scratch the Kauffman polynomial in Python. We start with a brief detour in computational knot theory and describe various representations of knots and links and find a good one to use for the algorithm. We then describe two approaches 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, finding an error for the knot 10_{125}.

Contents

1. Introduction

1.1. The Kauffman Polynomial

The Kauffman polynomial L is a two-variable polynomial invariant of regular isotopy for 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:

- $L\Big(\times\Big) + L\Big(\times\Big) = z\Big(L\Big(\asymp\Big) + L\Big(\,\big)\ \big(\,\Big)\Big)$

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

We will later be seeing that the Kauffman polynomial can be defined in a more explicit way, using a closed form recursive definition that we will be using to derive the first approach for our algorithm.

2. Computational Knot Theory

The first problem in computational knot theory is to find a good representation for knots and links. There are various 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 (S.G. 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 (P.D. 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.

- **D.T. (Dowker-Thistlethwaite) Codes** Is another representation present in KnotInfo based on crossing labels. We label each crossing twice in order, each crossing will get an even and odd label and we take only the even labels in order[1].

There are also other codes like **Braid representations** we will not be using in this project.

2.1. PD codes

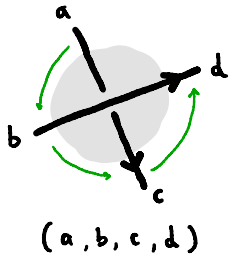
For our program we are going to use the P.D. code as input for our algorithm so let's explain how it is derived from a knot or link.

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 orientation. When we pass at a crossing that is an over-crossing for the current strand we write down

[1] Check this is correct

a 4-uple of counter-clockwise numbers for the arc incident to the crossing starting from the *entering under-crossing* arc.



Algorithm: The input is an oriented link diagram with starting points on each component and the output is a list of 4-tuples.

i) Choose an ordering for the components and starting point for each component

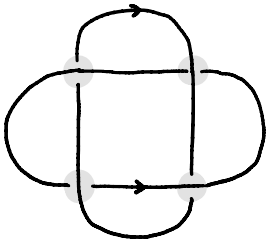
ii) Label each arc with a number

iii) For each component:

1) Walk along it from the starting point in its orientation

2) 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:

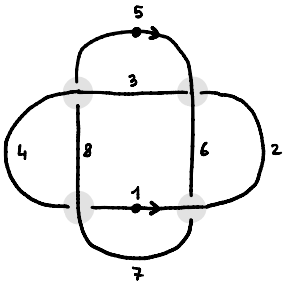
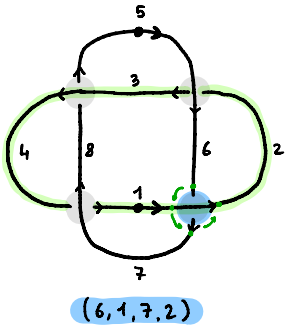
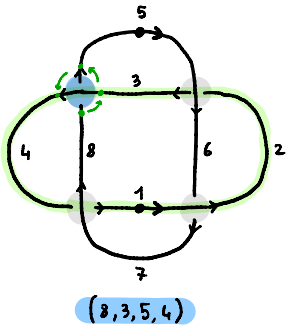


FIGURE 3. Oriented link with starting points and edge labels.

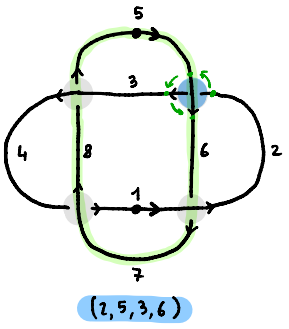
Now we can start processing each component of the link by walking along it in its orientation and writing down the over-crossings we encounter.



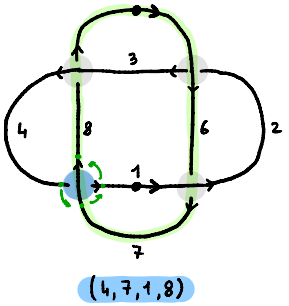
First link component
First over-crossing
⇒ `(6,1,7,2)`



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)]`

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**[2] 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

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

Algorithm: The input is an oriented link diagram with starting points on each component and the output is a list of components where each component is a list of pairs of numbers

i) Label each crossing with a number in order

ii) For each component:

   1) Walk along it from the starting point in its orientation

   2) At each crossing with label i, 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 fairly easy 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.

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** – To do this we need to swap the two over/under signs for the crossings for each of the two occurrences of the over and under strand. Then we also need to flip the handedness of the crossing as the switch causes an handedness change.
  - **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.

[2] Gauss Notation on Wikipedia

- **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.

    One can add a meaning to pairs (i, j) symbols to the original sequence of 4-tuples 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 do splices very often.

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, this would be ok for knots/links for up to 128 crossings), 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 $\approx 2n \times (N + 1.5) + k \times \lceil \log_2(n) \rceil$ bits of information. Each crossing appears twice and we store its id, over/under and handedness (each pair has the same handedness so we can store it only once per crossing) 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.

2.4. Link reconstruction from code

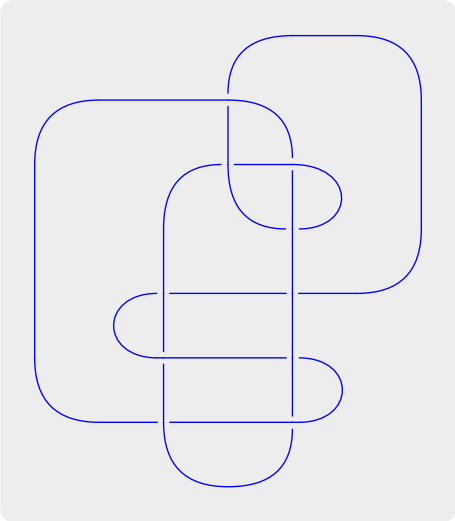
We briefly mention that reconstructing a link from a PD or SG code is not trivial and there are various approaches used by various softwares that can be used for this task.

2.4.1. 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 solved here is finding an orthogonal presentation for the link with the *minimum number of left and right bends*[3].

2.4.2. 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.

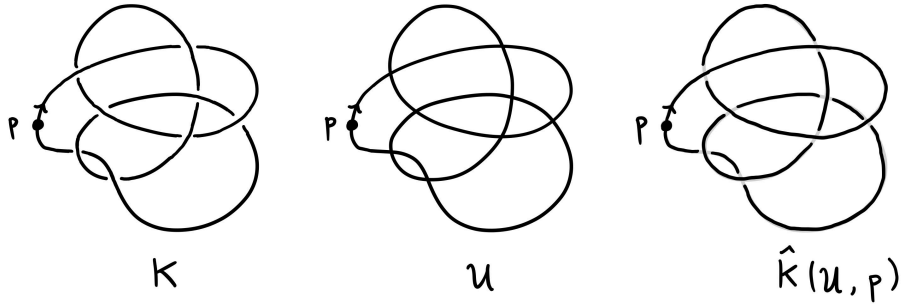
This condition that every point is the average of its neighbors can be easily expressed as a system of linear equations where some points on a chosen outer face have been fixed. When the graph is planar and 3-vertex-connected the linear system is non degenerate and has a unique solution.

3. Computing the Polynomial

We tried two approaches for our algorithm, the first based on the closed form algorithm present in Kauffman's paper and another "naive one" based directly on applying the skein relation. Let's first recap the main formal closed form algorithm for computing the Kauffman polynomial.

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

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*

3.0.1. First Approach

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

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

- $A_i^\lambda K \coloneqq E_i S_{\lambda_i} \cdots S_{\lambda_0} K$
- $B_i^\lambda K \coloneqq e_i S_{\lambda_i} \cdots S_{\lambda_0} K$
- 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

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

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

Closed Form 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

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

3.0.2. Second Approach

After a first implementation of the algorithm we noticed that it is not very efficient to compute using this closed form algorithm. In this case the naive implementation of the algorithm of just applying the rules recursively is more efficient and we only use the closed form for handling the base cases and the case of multiple components. The final algorithm we implemented is the following:

Algorithm: We apply the following rules recursively

i) If K is a standard unlink then return $a^{w(K)}$.

ii) If K has groups of components that overlap each other then we can apply the following rules, let $K = K_1 \sqcup \cdots \sqcup K_n$ be these groups of components one overling the other and let $d = (a + a^{-1})/z - 1$ then

$$L(K_1 \sqcup \cdots \sqcup K_n) = d^{n-1}L(K_1)\cdots L(K_n)$$

This is borrowed from the closed form algorithm and let's us solve the case of disjoint components.

iii) If K is a linked component then we find $\hat{K}$, the standard unlink for K, and pick the first available crossing c to switch, then we have some cases based on the crossing over/ under type and handedness that can be reduced to the following two cases

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

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

actually due to the symmetry of the Kauffman polynomial we can just use the same rule for both cases, so we can write

$$L_{K(a,z)} = z\Big(L_{E_cK}(a,z) + L_{e_cK}(a,z)\Big) - L_{S_cK}(a,z)$$

where E_cK and e_cK have one less crossing and S_cK has the same crossings as K but is one step closer to the standard unlink.

4. Python Implementation

The approach has been a mix of bottom-up and top-down, we first wrote the code for the main algorithm and then wrote the missing implementation for `SGCode` writing many tests along the way. First we defined a couple of classed `SGCode` and `PDCode` to work with these codes and easily convert between each other.

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.

```
class SGCodeCrossing:
    id: int
    over_under: typing.Literal[+1, -1]
    handedness: typing.Literal[+1, -1]

    def is_over(self) → bool:
    def is_under(self) → bool:
    def opposite(self) → SGCodeCrossing:
    def switch(self) → SGCodeCrossing:
    def flip_handedness(self) → SGCodeCrossing:
```

```
class SGCode:
    components: list[list[SignedGaussCodeCrossing]]

    def relabel(self) → SGCode:
    def to_minimal(self) → SGCode:

    def writhe(self) → int:
    def crossings_count(self) → int:

    def reverse(self, ids: list[int]):
    def mirror(self):

    def overlies_decomposition(self) → list[list[int]]:

    def std_unknot_switching_sequence(self) → list[int]:
    def apply_switching_sequence(self, seq: list[int]) → SGCode:
    def first_switch_to_std_unknot(self) → (int | bool):

    def sublink(self, component_ids: list[int]) → SGCode:

    def get_crossing_handedness(self, id: int) → Sign:
    def get_crossing_indices(self, id: int) → list[tuple[int, int, SGCodeCrossing]]:

    def switch_crossing(self, id: int) → SGCode:
    def splice_h(self, id: int, orthogonal: Sign = +1) → SGCode:
    def splice_v(self, id: int) → SGCode:
```

```
    # static methods
    def from_tuples(self, components: list[list[tuple[int, int]]]) → SignedGaussCode:
    def from_pd(pd_code: PDCode) → SGCode:
```

Let's now walk through the most important methods of this class.

4.1.1. 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.

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

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

4.1.2. 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 each component of the link. Then we can walk along the shadow of each component in order and make each crossing an over-crossing when passing on it on the first time.

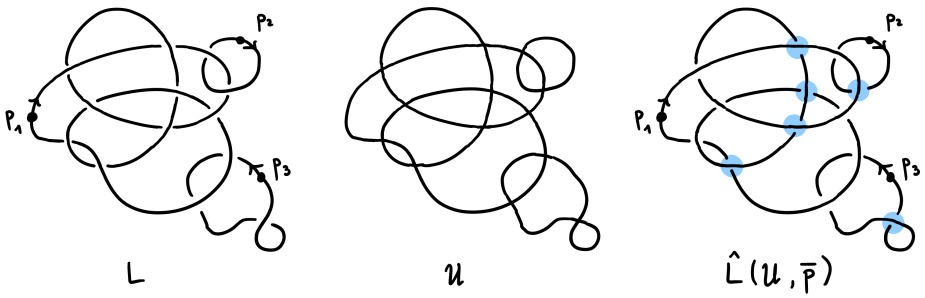


Figure 5. Standard unknot construction for a link, in blue are highlighted all the crossings in the resulting switching sequence

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

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 std_unknot_switching_sequence(self) -> list[int]:
    visited_crossings: set[int] = set() # list of ids
    switched_crossings: list[int] = [] # resulting list of ids to switch

    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
```

Now we need to be careful when applying these switches as they do not preserve the writhe as we can see in the following figure

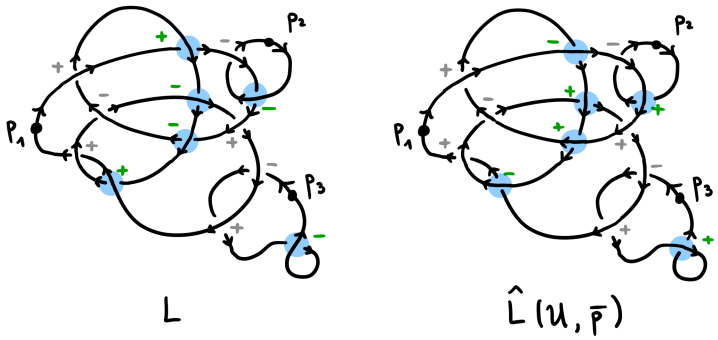


FIGURE 6. Switched crossing signs after brining the previous link to its standard unlink

First we wrote two methods, one called `SGCodeCrossing.switch()` that creates a new switched crossing from another crossing and then another one for `SGCode` that does the switches for the *two* occurrences.

```python
def switch(self: SGCodeCrossing):
    return SGCodeCrossing(
      self.id,
      -self.over_under,
      -self.handedness
    )
```

```python
def switch_crossing(self: SGCode, id: int):
    return SGCode([
        [
            crossing.switch()
            if crossing.id == id else crossing
            for crossing in component
        ]
        for component in self.components
    ])
```

With this infrastructure in place now applying a switching sequence is just a matter of applying all switches in a given sequence, this can be done in a single pass with the following function.

```python
def apply_switching_sequence(self, seq: list[int]) -> SGCode:
    return SGCode([
        [
            crossing.switch()
            if crossing.id in switching_sequence else crossing
            for crossing in component
        ]
```

```
            for component in self.components
        ])
```

In the actual Kauffman polynomial computation we just compute the first available switch in the switching sequence and apply just that one.

```
def first_switch_to_std_unknot(self) → (int | bool):
    visited_crossings: set[int] = set()
    for component in self.components:
        for crossing in component:
            if crossing.id not in visited_crossings and crossing.is_under():
                return crossing.id
            visited_crossings.add(crossing.id)
    return False
```

4.1.3. 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 components

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*.

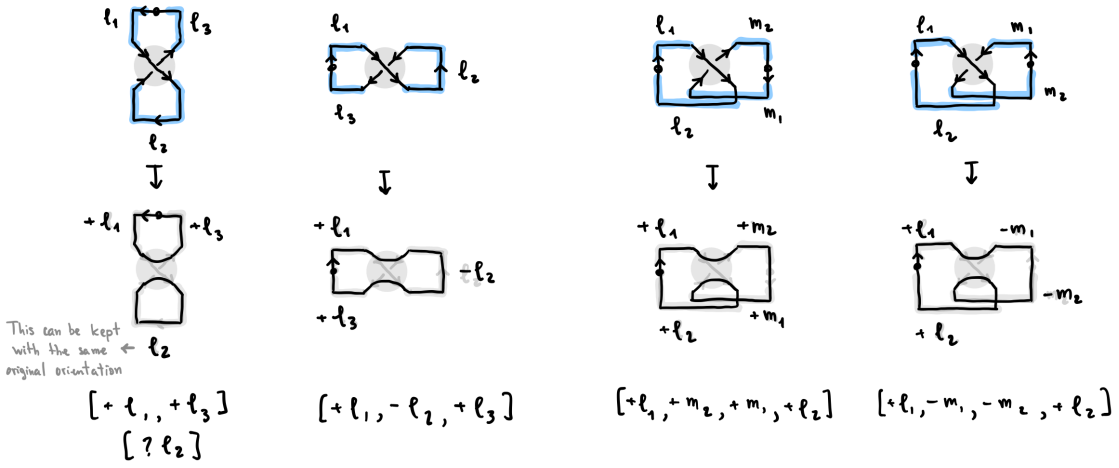


Figure 7. Cases for horizontal splicing

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 as showed in Figure 8.

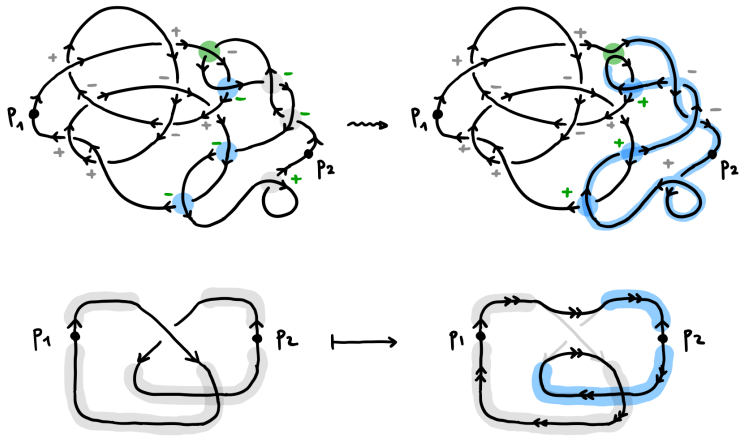


FIGURE 8. Crossing signs problem after splice

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. Let's see for example the code for the negative crossing horizontal splice case[4]

[4]The python operator `^` is the symmetry difference of sets

```
over_crossing_ids = set(c.id for c in l2 if c.is_over())
under_crossing_ids = set(c.id for c in l2 if c.is_under())
non_self_crossing_ids = over_crossing_ids ^ under_crossing_ids

def update_signs(strand: Iterable[SGCodeCrossing]):
    return [
        c.flip_handedness() if c.id in non_self_crossing_ids else c
        for c in strand
    ]

return SGCode([
    *(
        update_signs(component)
        for i, component in enumerate(self.components)
        if i != component_index
    ),
    [
        *update_signs(l1),
        *update_signs(reversed(l2)),
        *update_signs(l3),
    ],
])
```

The code for the **vertical splices** is omitted as the cases are the same as for the horizontal splices just flipped with respect to handedness. 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.

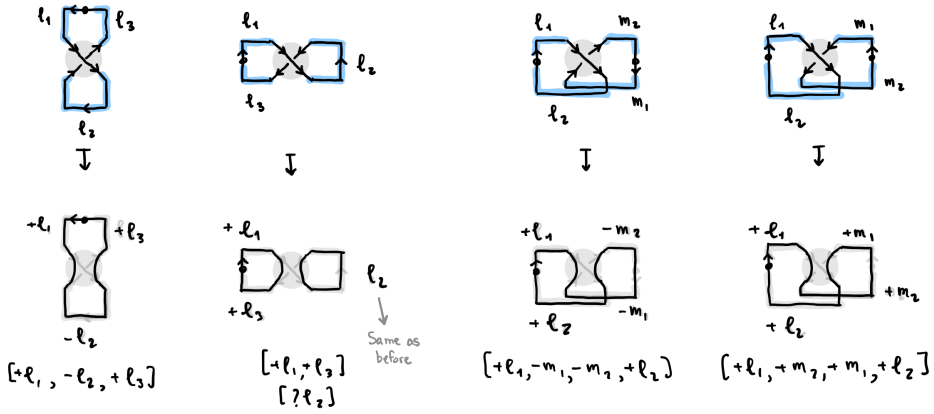


FIGURE 9. Cases for vertical splicing

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.

4.2. Code for computing the Kauffman polynomial

The final code for computing the Kauffman polynomial is the following, the main idea of the algorithm was already described previously. The code uses the `SGCode` class defined above to represent links. At the top level we define globals variables for `a`, `z` using the `sympy` library for working with polynomials.

```
a, z = symbols("a z")
d = (a + 1 / a) / z - 1

def kauffman_polynomial(link: SGCode) -> Poly:
    if len(link.components) == 0:
        return 0

    component_groups = link.overlies_decomposition()

    if len(component_groups) == 1:
        unknot_index = link.first_switch_to_std_unknot()

        link_rev = link.reverse()
        unknot_index_rev = link_rev.first_switch_to_std_unknot()

        if unknot_index == False or unknot_index_rev == False:
            return a ** link.writhe()
        else:
            link_switched = link.switch_crossing(unknot_index)
            link_spliced_h = link_switched.splice_h(unknot_index)
            link_spliced_v = link_switched.splice_v(unknot_index)

            k_link_spliced_h = kauffman_polynomial(link_spliced_h)
            k_link_spliced_v = kauffman_polynomial(link_spliced_v)
            k_link_switched = kauffman_polynomial(link_switched)

            return (
                z * (k_link_spliced_h + k_link_spliced_v)
                - k_link_switched
            )
    else:
        result = 1
        for k, component_ids in enumerate(component_groups):
            new_link = link.sublink(component_ids)
            if k > 0:
                result *= d

            result *= kauffman_polynomial(new_link)

        return result
```

Given this we can also write a function for computing the normalized Kauffman polynomial $F_{K(a,z)} = a^{-w(K)} L_{K(a,z)}$ as follows

```
def f_polynomial(link: SGCode) -> Poly:
    return (a ** (-link.writhe())) * kauffman_polynomial(link)
```

4.2.1. Debugging and Optimizations

The code (omitted from above, see the github repository for the full code) has actually been instrumented with some helper function to ease debugging and optimizations.

- Use the `cache` python decorator to memoize all calls to `kauffman_polynomial`.
- Another decorator called `log_input_output` helps with debug printing the traces like the following for the Hopf link

```
● kauffman_polynomial([[(+1, +1), (-2, +1)], [(+2, +1), (-1, +1)]])
│  [i] not cached...
│  [i] applying skein
│  [i] splice h, lambda = [2, ...]
│  ● kauffman_polynomial([[(+1, -1), (-1, -1)]])
│  │  [i] not cached...
│  │  [i] standard unknot form
│  └─► 1/a
│  [i] splice v, lambda = [2, ...]
│  ● kauffman_polynomial([[(+1, +1), (-1, +1)]])
│  │  [i] not cached...
│  │  [i] standard unknot form
│  └─► a
│  [i] switch, lambda = [2, ...]
│  ● kauffman_polynomial([[(+1, +1), (+2, -1)], [(-1, +1), (-2, -1)]])
│  │  [i] not cached...
│  │  [i] split link: [[0], [1]]
│  │  ● kauffman_polynomial([[]])
│  │  │  [i] not cached...
│  │  │  [i] standard unknot form
│  │  └─► 1
│  │  ● kauffman_polynomial([[]])
│  │  └─► 1
│  └─► a/z - 1 + 1/(a * z)
└─► a * z - a/z + 1 + z/a - 1/(a * z)
```

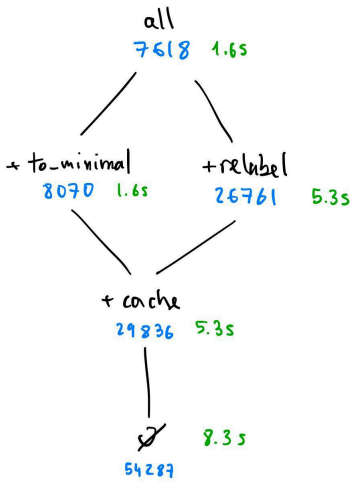
- Finally there is another decorator called `polynomial_wrapper` that help with applying optimizations to function calls and is defined as follows

```python
OptimizationType = Literal['expand', 'relabel', 'to_minimal']

def polynomial_wrapper(optimizations: set[OptimizationType] = {'expand'}):
    def decorator(func: Callable[[SGCode], Poly]) -> Callable[[SGCode], Poly]:
        @functools.wraps(func)
        def wrapper(link: SGCode) -> Poly:
            # First we convert to minimal rotated form and only then we relabel,
            # this ensures a consistent indexing for the cache.
            if 'to_minimal' in optimizations:
                link = link.to_minimal()
            if 'relabel' in optimizations:
                link = link.relabel()

            result = func(link)

            if 'expand' in optimizations:
                result = sympy.expand(result)

            return result
        return wrapper
    return decorator
```

This applies `sympy.expand(...)` to keep the resulting polynomial simplified and prepends each call with the two following optimizations

- `to_minimal`: This "rotates" the list of each component in the SG code, brining each list in *minimal lexicographical order*.
- `relabel`: After minimal rotation we apply a relabelling to increase the chances of hitting the cache decorator.

These two optimization decrease the number of calls by almost an order of magnitude, for example for the case of knot `12n_888` we have the following optimizations lattice, the number of total recursive calls for the function `kauffman_polynomial` is in blue and time of execution in green.



As we can see the most important optimization is the `to_minimal` one that by its own reduces the total number of calls from 29k to 8k but relabelling is still able to help.

All optimizations are enabled by default in the final program and debugging traces are disabled to not impact the speed.

4.3. Experiments

The main achievement of this project was to check all (normalized) Kauffman polynomials present in the KnotInfo database. Using the python package `database_knotinfo` we loaded all the data and wrote a script called `./check_knotinfo.py` with the following options

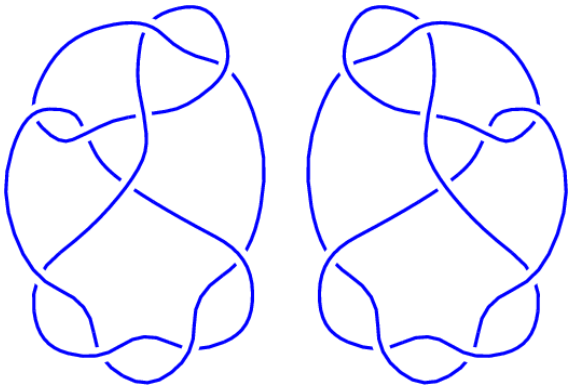
```
usage: check_knotinfo.py [-h] [--knots] [--links] [-c COUNT]

Calculate the Kauffman polynomial of a knot or link.

options:
  -h, --help            show this help message and exit
  --knots               Include knots from database
  --links               Include links from database
  -c, --count COUNT     Number of knots to test per database
```

This uses the `multiprocessing` library to parallelize the computation of the Kauffman polynomial for each call of the Kauffman polynomial and checks all the databases of knots and links in a couple of minutes.

We run this on all knots and links in the database and found the following results:

- For **links**: All 4187 include the Kauffman polynomial and are all correctly computed by our algorithm.
- For **knots**: Out of 12965 knots only 2977 knots include the Kauffman polynomial (they are computed only up to 12 crossing even if there are up to 13 crossings knots in the database) and *we found a mismatch in the computation of the polynomial of* 10_{125}.

4.3.1. The knot 10_{125}



This knot is *chiral* meaning that it is not equivalent to its mirror. Our algorithm computes the following polynomial for this knot

$$\begin{aligned} F_{10_{125}}(a,z) = {} & z^8\left(\frac{1}{a^2}+1\right) + z^7\left(a+\frac{2}{a}+\frac{1}{a^3}\right) + z^6\left(-6-\frac{6}{a^2}\right) \\ & + z^5\left(-5a-\frac{11}{a}-\frac{6}{a^3}\right) + z^4\left(2a^2+13+\frac{11}{a^2}\right) \\ & + z^3\left(a^3+8a+\frac{17}{a}+\frac{10}{a^3}\right) + z^2\left(a^4-6a^2-15-\frac{8}{a^2}\right) \\ & + z\left(a^5-a^3-6a-\frac{8}{a}-\frac{4}{a^3}\right) + 3a^2+7+\frac{3}{a^2} \end{aligned}$$

while the polynomial for the knot present in KnotInfo is the following

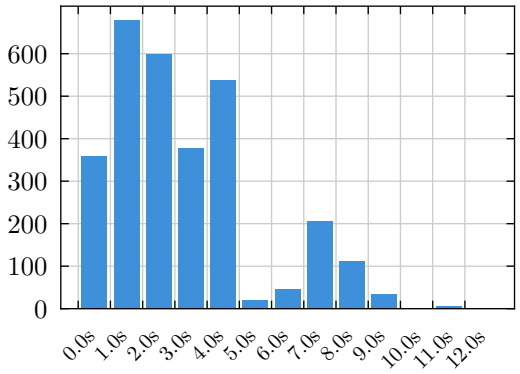
$$\begin{aligned} F'_{10_{125}}(a,z) = {} & z^8(a^2+1) + z^7\left(a^3+2a+\frac{1}{a}\right) + z^6(-6a^2-6) \\ & + z^5\left(-6a^3-11a-\frac{5}{a}\right) + z^4\left(11a^2+13+\frac{2}{a^2}\right) \\ & + z^3\left(10a^3+17a+\frac{8}{a}+\frac{1}{a^3}\right) + z^2\left(-8a^2-15-\frac{6}{a^2}+\frac{1}{a^4}\right) \\ & + z\left(-4a^3-8a-\frac{6}{a}-\frac{1}{a^3}+\frac{1}{a^5}\right) + 3a^2+7+\frac{3}{a^2} \end{aligned}$$

Then we noticed that they are related by the substitution $(a \mapsto 1/a, z \mapsto z)$. The Kauffman polynomial has this property that inverts the a when computing the mirror image of a knot, in this sense it is able to distinguish chiral variants of knots.

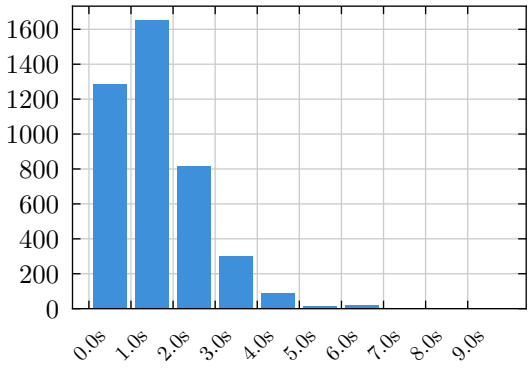
So, we checked using the implementation of the Kauffman polynomial using one in the `KnotTheory` Mathematica package: using the PD code from the KnotInfo database, both Mathematica and our algorithm give the **same result**. This makes us believe that there is a mismatch in KnotInfo between the PD code and Kauffman polynomial columns, meaning they are not correctly synchronized.

4.3.2. Performance Analysis

We also checked the performance of our algorithm on all knots and links in the database up to 12 crossings (these are the ones with the Kauffman polynomial in the database). The results are shown in the following histograms, each bar counts the number of knots that took the amount of time in the relative bin.



Histogram of knots times



Histogram of links times

We can see that the times are just about of a couple of seconds per knot or link. This could be improved by using more sophisticated caching techniques but we did not implement this as the performance was already good enough for our purposes.

5. Conclusion

In this project we implemented the Kauffman polynomial from scratch in Python using signed Gauss codes. We compared the results of our algorithm with the ones present in the KnotInfo database and found an error in the computation of the polynomial of the knot 10_{125}. We believe that the error is due to a mismatch between the PD code stored in the database with the corresponding Kauffman polynomial.