Arrow Polynomial Collapses for Almost-Classical Virtual Links

Levi Keck & Jason Tu (Ohio State University)

Overview

Core Definitions

Alexander numbering: Given an oriented virtual link, assign an integer to every arc (section of the link between two classical crossings) so that there exists some integer *i* such that when each classical crossing is rotated so that both outward strands face to the right, the labels go from *i* to $i + 1$ from bottom left to top right and from $i + 1$ to *i* from top left to bottom right. If a virtual link has a presentation which admits an Alexander numbering, it is called almost classical.

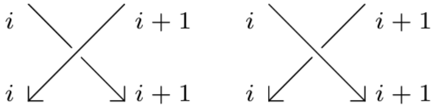
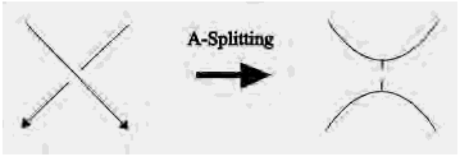


Figure: Alexander Numbering

State Expansion: choose an A- or B-oriented smoothing at every classical crossing. When there is no induced orientation, each of the resulting pieces is given a pole which points inwards towards the location where the crossing previously existed. The result is a collection of oriented loops with poles.



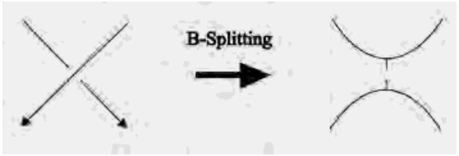


Figure: Creation of Poles in State Expansion

Loop index: For a loop L of state S, an index $\imath$ is assigned by moving all poles past virtual crossings onto a small semi-arc before canceling all adjacent poles on the same side of a loop. The index is then given by $\imath(L) = \frac{\#\text{poles}}{2}$.

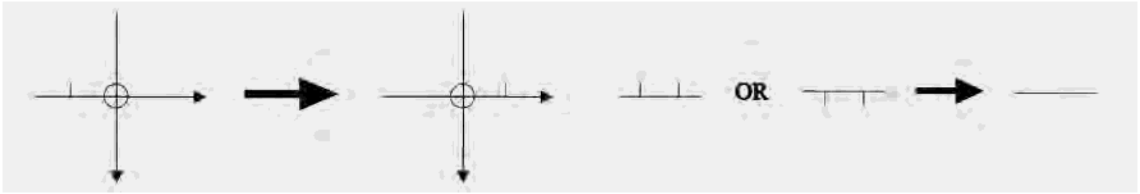


Figure: Rules for Pole Cancellation

Arrow Polynomial R_D: After one evaluates all states of a virtual link D and finds the index for each, the normalized Dye-Kauffman arrow polynomial R_D is given by

$$R_D(A; K_1, K_2, ...) = (-A^3)^{-w(d)} \sum_S A^{\alpha(S)-\beta(S)}(-A^2 - A^{-2})^{\delta(S)-1} \prod_{L \in S} K_{\imath(L)}$$

where $\alpha(S)$ is the number of A-splittings performed to reach state S, $\beta(S)$ is the number of B-splittings performed to reach state S, and $\delta(S)$ is the number of loops in S.

Main Theorem

Theorem

Let D be a virtual link diagram that admits an integral Alexander numbering (i.e. D is almost classical). Then every state in the arrow-polynomial expansion has total loop index 0*. Hence*

$$R_D\big(A\,;K_1,K_2,\dots\big)=R_D(A)\in\mathbb{Z}[A^{\pm1}].$$

Proof

We first examine what the state expansion looks like for a crossing with Alexander numbered arcs.

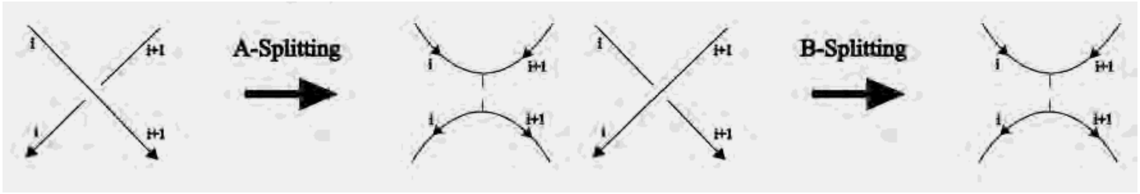


Figure: Crossings Expanded with Alexander Numbering

In either case of pole creation, we have one of the following two cases locally around each pole:

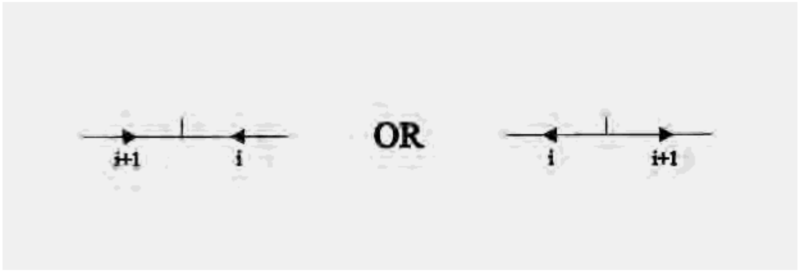


Figure: Local Picture Around Pole After Expansion of Crossing

We will call the first image a pole of type A and the second image a pole of type B.

Each loop has an even number of poles, so any pole will always be adjacent to at least one other pole. A pole of type A cannot be next to another pole of type A since the arrows on a pole of type A point towards the pole, so the arrow between them would have to point to both simultaneously. Similarly, two poles of type B cannot be adjacent. Below are the two possible configurations for poles of type A and B to be adjacent:

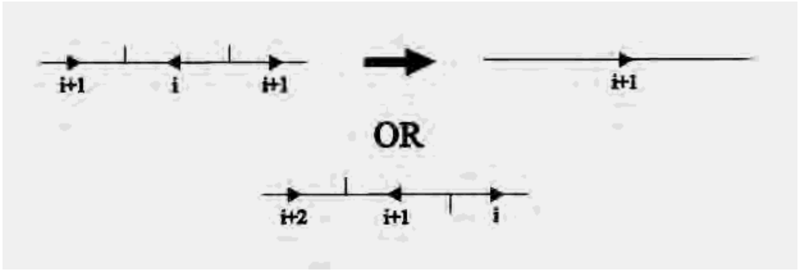


Figure: Possibilities for Adjacent Poles

After ignoring all pairs of the first type, the poles on a loop can all be grouped into pairs of the second type, with some potentially upside down with reconfigured indices. However, as seen below, a pair of the second type next to a pair of the second type flipped cancel each other out:

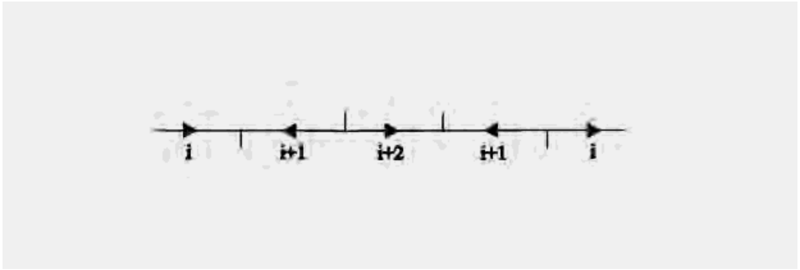


Figure: Two Pairs of the Second Type with One Flipped

After repeating the process of canceling out all adjacent pairs of the second type with one pair flipped and the other not, we will be left with a collection of pole pairs of the second type where every pair is flipped or every pair is not flipped. If you travel along the loop in the correct direction (which one depends on whether the poles are flipped or not), the label will increase by two across every pair. However, if there is at least one pair, this means that the indices will continue to increase indefinitely as you traverse the loop in that direction. However, assuming the original link had only finitely many crossings, there can only be finitely many labels on the loop, leading to a contradiction. Thus, the loop must have no poles and have index zero.

Consequences

First, we note that every classical link admits an Alexander numbering, and thus has an arrow polynomial with no "extra variables". To see this, we introduce the concept of the winding number. The winding number of a curve around a point is the signed number of counterclockwise rotations the curve makes around the point (clockwise rotations are counted as negative).

Importantly, each point in a region with no arcs has the same winding number and moving across an arc from right to left increases the winding number by 1.

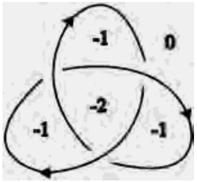


Figure: Trefoil with Winding Numbers

Now, for each arc, assign a label equal to the minimum of the two winding numbers of the regions it borders. This is well-defined, and around each crossing the labels look like the following, agreeing with the conditions for an Alexander numbering.

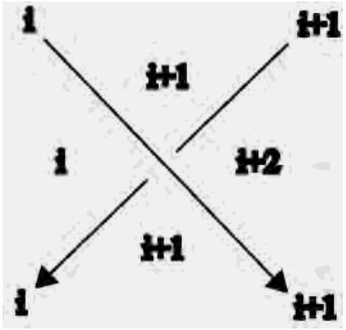


Figure: Alexander Numbering Induced by Winding Numbers

The next consequence is that by computing the arrow polynomial of certain virtual links, we can detect extra variables and conclude that no Alexander numbering is possible. For example, the following virtual link (4.26) has arrow polynomial $1 + A^{-2}K_3 - A^{-2}K_1K_2 + A^2K_1 - A^2K_1K_2$, meaning no Alexander numbering is possible.

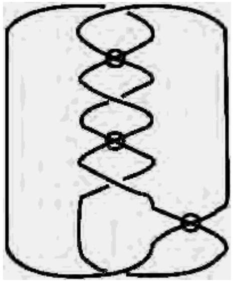


Figure: Virtual Knot 4.26

In this case, it is easy to verify that the result is correct after giving the graph an orientation and choosing labels for an initial crossing as in the diagram below. However, it could be difficult to disprove the existence of an Alexander numbering for an equivalent presentation with more crossings. The arrow polynomial guarantees that none will be Alexander numberable.

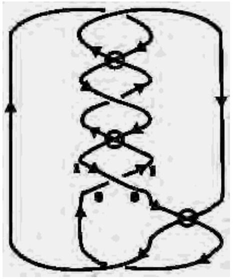


Figure: Virtual Knot 4.26 Oriented

Example: classical vs. virtual

Classical trefoil

- Integral Alexander numbering exists.
- Arrow polynomial reduces to a Laurent polynomial in $\mathbb{A}$: $R_{\text{cl.trefoil}}(\mathbb{A}) \in \mathbb{Z}[A^{\pm 1}]$.

Virtual trefoil

- No integral numbering.
- A state with four like-oriented cusps yields factor K_2.

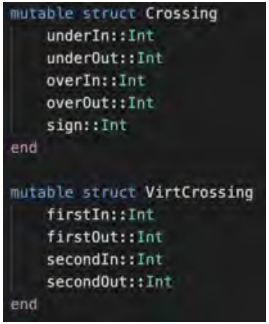
Beyond almost-classical

- **Mod-*m* numberings**: The same argument can conclude that if m is odd only $K_m, K_{2m}, \ldots$ may appear in the arrow polynomial, and if m is even only $K_{\frac{m}{2}}$, K_m, $K_{\frac{3m}{2}}$, ... may appear.
- **Twisted links** (Deng '22): checkerboard-colourability in a non-orientable surface also forces total cusp-index 0.
- Open question: characterize the largest class of virtual links whose arrow polynomial is single-variable.

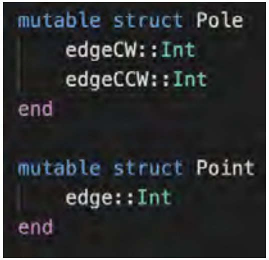
Take-home message

- Almost-classical links behave "classically" as far as the arrow polynomial is concerned.
- Direct state-sum proof relies only on Alexander numbering + simple counting.
- Generalizations hint at a deeper relation between parity/colourability and arrow-variable suppression.

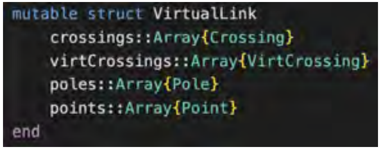
Arrow Polynomial Program



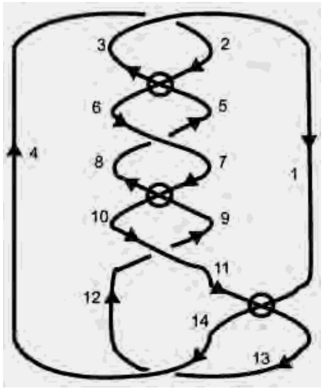
```
mutable struct Crossing
    underIn::Int
    underOut::Int
    overIn::Int
    overOut::Int
    sign::Int
end

mutable struct VirtCrossing
    firstIn::Int
    firstOut::Int
    secondIn::Int
    secondOut::Int
end
```



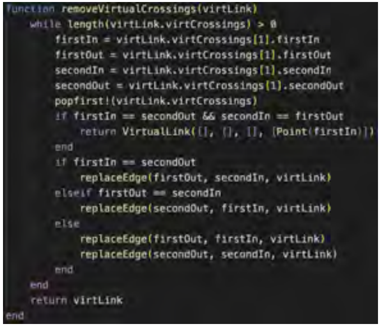
```
mutable struct Pole
    edgeCW::Int
    edgeCCW::Int
end

mutable struct Point
    edge::Int
end
```



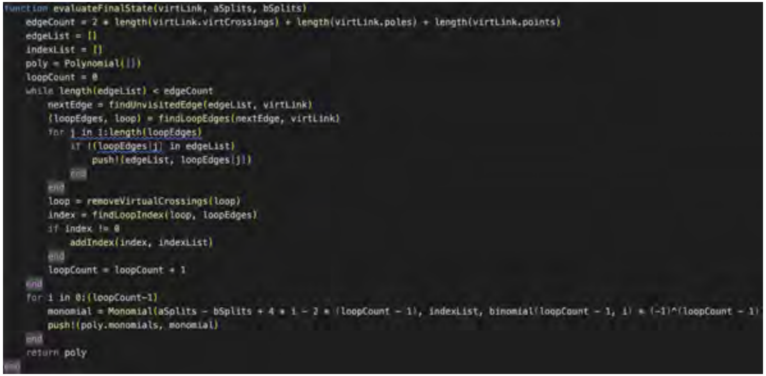
```
mutable struct VirtualLink
    crossings::Array{Crossing}
    virtCrossings::Array{VirtCrossing}
    poles::Array{Pole}
    points::Array{Point}
end
```



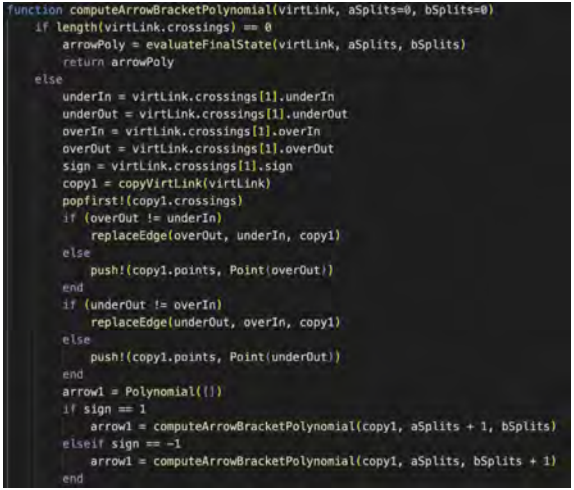
```
link = VirtualLink([Crossing(4, 2, 3, 1, -1), Crossing(8, 5, 6, 7, 1), Crossing(12, 9, 10, 11, 1), Crossing(13, 12, 14, 4, -1)], [VirtCrossing(2, 6, 5, 3), VirtCrossing(7, 10, 9, 8), VirtCrossing(11, 13, 1, 14)], [], [])
```



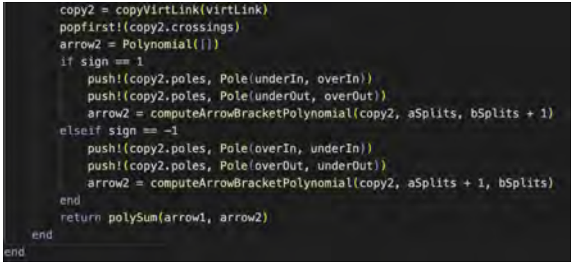
```
function removeVirtualCrossings(virtLink)
    while length(virtLink.virtCrossings) > 0
        firstIn = virtLink.virtCrossings[1].firstIn
        firstOut = virtLink.virtCrossings[1].firstOut
        secondIn = virtLink.virtCrossings[1].secondIn
        secondOut = virtLink.virtCrossings[1].secondOut
        popfirst!(virtLink.virtCrossings)
        if firstIn == secondOut && secondIn == firstOut
            return VirtualLink([], [], [], [Point(firstIn)])
        end
        if firstIn == secondOut
            replaceEdge(firstOut, secondIn, virtLink)
        elseif firstOut == secondIn
            replaceEdge(secondOut, firstIn, virtLink)
        else
            replaceEdge(firstOut, firstIn, virtLink)
            replaceEdge(secondOut, secondIn, virtLink)
        end
    end
    return virtLink
end
```



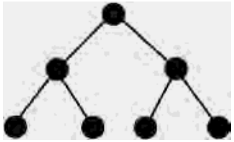
```
function evaluateFinalState(virtLink, aSplits, bSplits)
    edgeCount = 2 * length(virtLink.virtCrossings) + length(virtLink.poles) + length(virtLink.points)
    edgeList = []
    indexList = []
    poly = Polynomial([])
    loopCount = 0
    while length(edgeList) < edgeCount
        nextEdge = findUnvisitedEdge(edgeList, virtLink)
        (loopEdges, loop) = findLoopEdges(nextEdge, virtLink)
        for j in 1:length(loopEdges)
            if !(loopEdges[j] in edgeList)
                push!(edgeList, loopEdges[j])
            end
        end
        loop = removeVirtualCrossings(loop)
        index = findLoopIndex(loop, loopEdges)
        if index != 0
            addIndex(index, indexList)
        end
        loopCount = loopCount + 1
    end
    for i in 0:(loopCount-1)
        monomial = Monomial(aSplits - bSplits + 4 * i - 2 * (loopCount - 1), indexList, binomial(loopCount - 1, i) * (-1)^(loopCount - 1)
        push!(poly.monomials, monomial)
    end
    return poly
end
```



```
function computeArrowBracketPolynomial(virtLink, aSplits=0, bSplits=0)
    if length(virtLink.crossings) == 0
        arrowPoly = evaluateFinalState(virtLink, aSplits, bSplits)
        return arrowPoly
    else
        underIn = virtLink.crossings[1].underIn
        underOut = virtLink.crossings[1].underOut
        overIn = virtLink.crossings[1].overIn
        overOut = virtLink.crossings[1].overOut
        sign = virtLink.crossings[1].sign
        copy1 = copyVirtLink(virtLink)
        popfirst!(copy1.crossings)
        if (overOut != underIn)
            replaceEdge(overOut, underIn, copy1)
        else
            push!(copy1.points, Point(overOut))
        end
        if (underOut != overIn)
            replaceEdge(underOut, overIn, copy1)
        else
            push!(copy1.points, Point(underOut))
        end
        arrow1 = Polynomial([])
        if sign == 1
            arrow1 = computeArrowBracketPolynomial(copy1, aSplits + 1, bSplits)
        elseif sign == -1
            arrow1 = computeArrowBracketPolynomial(copy1, aSplits, bSplits + 1)
        end
```



```
        copy2 = copyVirtLink(virtLink)
        popfirst!(copy2.crossings)
        arrow2 = Polynomial([])
        if sign == 1
            push!(copy2.poles, Pole(underIn, overIn))
            push!(copy2.poles, Pole(underOut, overOut))
            arrow2 = computeArrowBracketPolynomial(copy2, aSplits, bSplits + 1)
        elseif sign == -1
            push!(copy2.poles, Pole(overIn, underIn))
            push!(copy2.poles, Pole(overOut, underOut))
            arrow2 = computeArrowBracketPolynomial(copy2, aSplits + 1, bSplits)
        end
        return polySum(arrow1, arrow2)
    end
end
```



Key references

N. Kamada, *A multivariable polynomial invariant of virtual links and cut systems*, 2021.

A. Nakamura, Y. Nakanishi, A. Satoh, and S. Tomiyama, *Twin groups and their applications in virtual knot theory*, JKTR (2012).

W. Deng, *Arrow polynomial of twisted links*, JKTR (2022).