# Arrow Polynomial from a Gauss Code

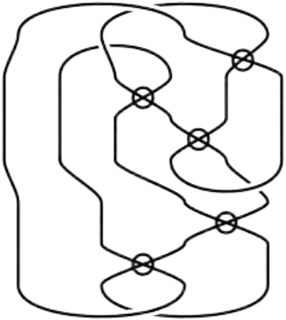
Mark Kikta

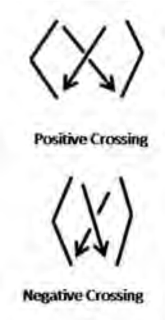
# Gauss Code

## Definition

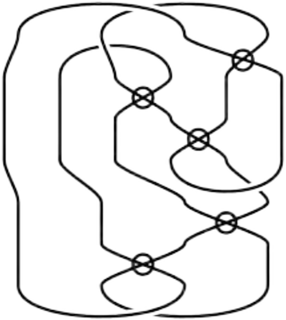
A **Gauss code** is a string that uniquely determines an underlying virtual knot. It may be obtained from a virtual knot diagram by fixing a starting point somewhere along then knot, fixing an orientation, and labelling the classical crossings of the diagram; then travelling along the diagram with the orientation and recording whether the strand we are on is the over- or under-crossing strand, the label of the crossing, and the sign of the crossing.

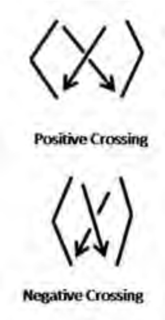
# Example





# Example





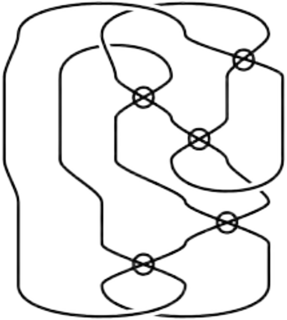

O1-O2+U1-O3-U2+U4+U3-O4+

# Planar Diagrams

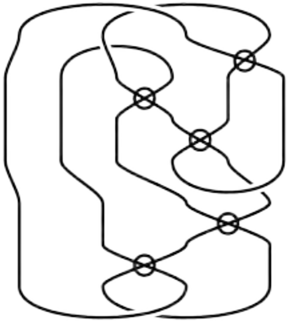
## Definition

A **signed planar diagram** is a list of crossing information that uniquely determines an underlying virtual knot. It may be obtained from a virtual knot diagram by labelling the arcs and, for each crossing, recording its sign and the arcs meeting at the crossing, starting with the label on the tail of the over-crossing arc and proceeding clockwise (although this is not always the convention).

# Example



# Example

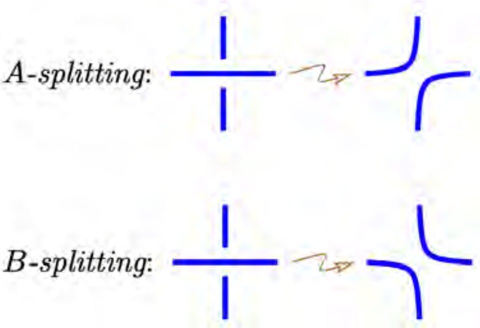


$$[+|8,4,7,5],[-|1,7,8,6],[+|2,3,1,4],[-|6,3,5,2]$$

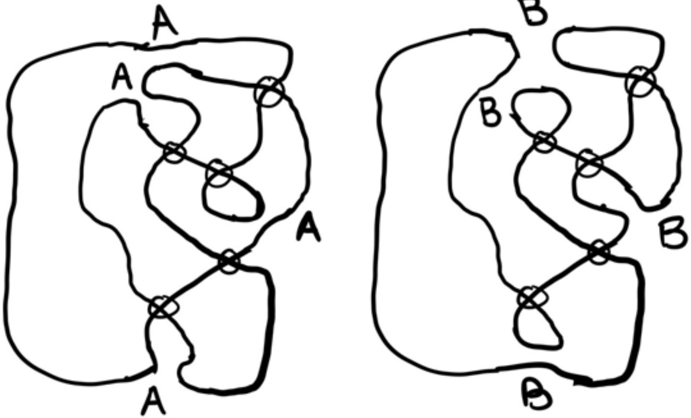
# States of a Diagram

## Definition

A **state** of a diagram is a choice of an **A-splitting** or **B-splitting** at every crossing.



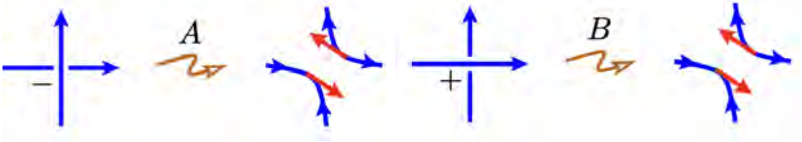
# Some States of 4_19



# Arrow Polynomial

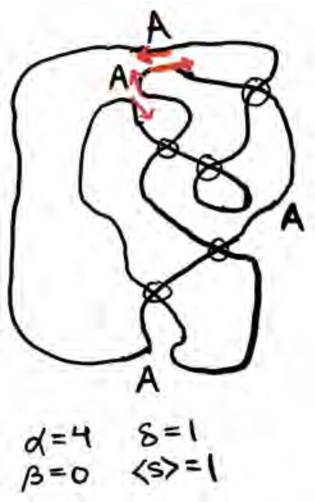


Let $L$ be an oriented diagram. If an arc splitting disagrees with the orientation on $L$, put an arrow on each arc of the splitting, oriented counterclockwise.




Then on each state, remove all sets of two adjacent arrows pointing in the same direction.

# Arrow Polynomial



# Arrow Polynomial

## Arrow Polynomial

The **arrow polynomial** of a knot $K$ is

$$[K]_A(A, B, d, k_i) = \sum_{s \in S} A^{\alpha(s)} B^{\beta(s)} d^{\delta(s)-1} \langle s \rangle,$$

where $i(c)$ = half the number of remaining arrows on the circle $c$, $\langle s \rangle = \prod_{c \in s} k_{i(c)}$, $\alpha(s)$ = #A-splittings in $s$, $\beta(s)$ = #B-splittings in $s$, and $\delta(s)$ = #circles in $s$.

## Normalized Arrow Polynomial

$$\langle K \rangle_A = [L]_A(A, A^{-1}, (-A^2 - A^{-2}), k_i)$$

## Gauss Code to Planar Diagram



We need a data structure for Gauss codes and a data structure for planar diagrams:

- We represent a Gauss code as a list of tuples $(isOver, label, sign)$, where $isOver$ is true iff the strand we are on is an over-crossing strand, $label$ is the label of the crossing, and $sign$ is true iff the crossing is positive.

# Gauss Code to Planar Diagram



We need a data structure for Gauss codes and a data structure for planar diagrams:

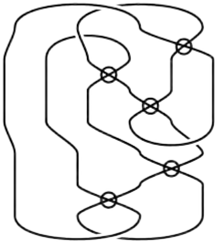
- We represent a Gauss code as a list of tuples $(isOver, label, sign)$, where *isOver* is true iff the strand we are on is an over-crossing strand, *label* is the label of the crossing, and *sign* is true iff the crossing is positive.
- We represent a planar diagram as a list of tuples $(sign, labels)$, where *sign* is true iff the crossing is positive and *labels* is an ordered list of the labels on the arcs around the crossing.

# Gauss Code to Planar Diagram

For each code in a Gauss code:

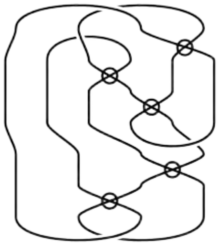
1. If this label has not yet been encountered, add a new crossing to the planar diagram. Record its sign and label the arcs on the strand we are travelling along.
2. If this label has already been encountered, add labels to the strand that has not yet been travelled along.

# Example



O1-O2+U1-O3-U2+U4+U3-O4+

# Example



O1-O2+U1-O3-U2+U4+U3-O4+

$[+|8,4,7,5],[-|1,7,8,6],[+|2,3,1,4],[-|6,3,5,2]$

# Pseudocode

**function** GAUSSCODETOPLANARDIAGRAM($gaussCode$)
  $crossings \leftarrow \{\}$
  $arcNumber \leftarrow 0$
  **for** $code$ in $gaussCode$ **do**
    $nextArcNumber \leftarrow (arcNumber + 1)\%|gaussCode|$
    **if** $code.label$ not in $crossings$ **then**
      **if** $code.isOver$ **then**
        $crossings[code.label] \leftarrow crossing(code.sign, nextArcNumber, None, arcNumber, None)$
      **else if** $code.sign$ **then**
        $crossings[code.label] \leftarrow crossing(code.sign, None, arcNumber, None, nextArcNumber)$
      **else** $crossings[code.label] \leftarrow crossing(code.sign, None, nextArcNumber, None, arcNumber)$
      **end if**
    **else**
      **if** $code.isOver$ **then**
        $crossings[code.label].labels[0] \leftarrow nextArcNumber$
        $crossings[code.label.labels[2] \leftarrow arcNumber$
      **else if** $code.sign$ **then**
        $crossings[code.label].labels[1] \leftarrow arcNumber$
        $crossings[code.label.labels[3] \leftarrow nextArcNumber$
      **else**
        $crossings[code.label].labels[1] \leftarrow nextArcNumber$
        $crossings[code.label.labels[3] \leftarrow arcNumber$
      **end if**
    **end if**
    $arcNumber \leftarrow arcNumber + 1$
  **end forreturn** $crossings$
**end function**

# Crossing Expansion

We need a data structure to represent an arc created from a splitting and a data structure to represent a state with arrows:

- The two arcs created from a splitting will be represented as tuples $(\mathit{isOriented}, \mathit{labelOne}, \mathit{labelTwo})$, where *isOriented* is true iff the arc has an arrow and labels one and two are inherited from the labels on the crossing.

# Crossing Expansion

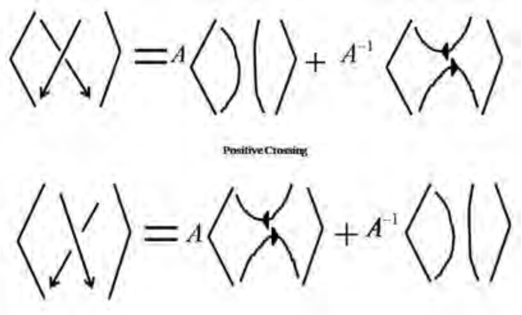


We need a data structure to represent an arc created from a splitting and a data structure to represent a state with arrows:

- The two arcs created from a splitting will be represented as tuples $(isOriented, labelOne, labelTwo)$, where $isOriented$ is true iff the arc has an arrow and labels one and two are inherited from the labels on the crossing.
- We represent a state as a tuple $(weight, arcs)$, where $weight$ is the power of $A$ associated with the state and $arcs$ is a list of arcs in the state.

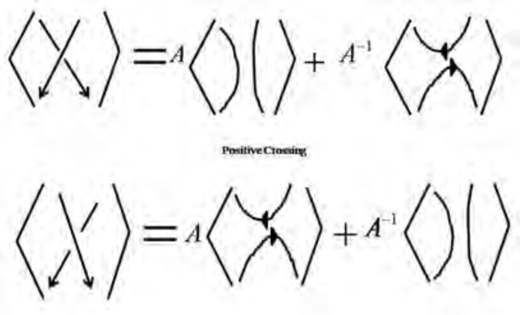
# Crossing Expansion

We can recursively find each state by expanding crossings and updating the weight of the state at each step.



# Crossing Expansion

We can recursively find each state by expanding crossings and updating the weight of the state at each step.




$$[True, 0, 1, 2, 3] = A[(False, 1, 0), (False, 2, 3)] + A^{-1}[(True, 2, 1), (False, 0, 3)]$$
$$[False, 0, 1, 2, 3] = A[(True, 3, 2), (True, 1, 0)] + A^{-1}[(False, 2, 1), (False, 3, 0)]$$

# Pseudocode

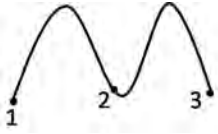
```
function EXPANDCROSSINGS(state)
    states ← []
    function EXPANDCROSSING(i, state)
        if i = |state| then
            append state to states return
        end if
        crossing ← state[i]
        branchOne ← copy of state
        branchTwo ← copy of state
        if crossing.sign then
            branchOne.weight ← branchOne.weight ∗ A
            append arc(False, crossing.labels[1], crossing.labels[0]) to branchOne
            append arc(False, crossing.labels[2], crossing.labels[3]) to branchOne
            branchTwo.weight ← branchTwo.weight/A
            append arc(True, crossing.labels[2], crossing.labels[1]) to branchOne
            append arc(True, crossing.labels[0], crossing.labels[3]) to branchOne
        else
            branchOne.weight ← branchOne.weight ∗ A
            append arc(True, crossing.labels[3], crossing.labels[2]) to branchOne
            append arc(True, crossing.labels[1], crossing.labels[0]) to branchOne
            branchTwo.weight ← branchTwo.weight/A
            append arc(False, crossing.labels[2], crossing.labels[1]) to branchOne
            append arc(False, crossing.labels[3], crossing.labels[0]) to branchOne
        end if
        expandCrossing(i + 1, branchOne)
        expandCrossing(i + 1, branchTwo)
    end function
    expandCrossing(0, state(1, []))
end function
```

# State Reduction

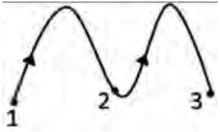
We need to cancel adjacent arrows oriented in the same direction, and we can reduce the number of arcs in the process to simplify further calculation. There are five possible cases:

# State Reduction

We need to cancel adjacent arrows oriented in the same direction, and we can reduce the number of arcs in the process to simplify further calculation. There are five possible cases:




$$[(\mathit{False}, 1, 2), (\mathit{False}, 2, 3)] \to (\mathit{False}, 1, 3)$$

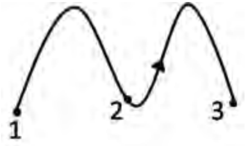



$$[(\mathit{True}, 1, 2), (\mathit{True}, 2, 3)] \to (\mathit{False}, 1, 3)$$

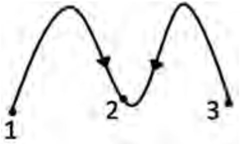
# State Reduction




$$[(\mathit{True}, 1, 2), (\mathit{False}, 2, 3)] \to (\mathit{True}, 1, 3)$$




$$[(\mathit{False}, 1, 2), (\mathit{True}, 2, 3)] \to (\mathit{True}, 1, 3)$$




Cannot be reduced!

# Pseudocode I



```
function REDUCESTATE(state)
    function FINDREDUCTION
        for i = 0 to |state| do
            for j = 0 to |state| do
                if i ≠ j then
                    if not state[i].isOriented and not state[j].isOriented and state[i].labelTwo = state[j].labelOne then
                        state[i].labelTwo ← state[j].labelTwo
                        state.pop(j) return True
                    else if not state[i].isOriented and not state[j].isOriented and state[i].labelTwo = state[j].labelTwo then
                        state[i].labelTwo ← state[j].labelOne
                        state.pop(j) return True
                    else if state[i].isOriented and state[j].isOriented and state[i].labelTwo = state[j].labelOne then
                        state[i].labelTwo ← state[j].labelTwo
                        state[i].isOriented ← False
                        state.pop(j) return True
                    else if state[i].isOriented and not state[j].isOriented and state[i].labelTwo = state[j].labelOne then
                        state[i].labelTwo ← state[j].labelTwo
                        state.pop(j) return True
                    else if state[i].isOriented and not state[j].isOriented and state[i].labelTwo = state[j].labelTwo then
                        state[i].labelTwo ← state[j].labelOne
                        state.pop(j) return True
                    else if state[i].isOriented and not state[j].isOriented and state[i].labelOne = state[j].labelTwo then
                        state[i].labelOne ← state[j].labelTwo
                        state.pop(j) return True
                    else if state[i].isOriented and not state[j].isOriented and state[i].labelOne = state[j].labelOne then
                        state[i].labelOne ← state[j].labelTwo
                        state.pop(j) return True
```

# Pseudocode II



```
                end if
            end if
        end for
    end for
end function
while findReduction do
    nothing
end while
```
**end function**

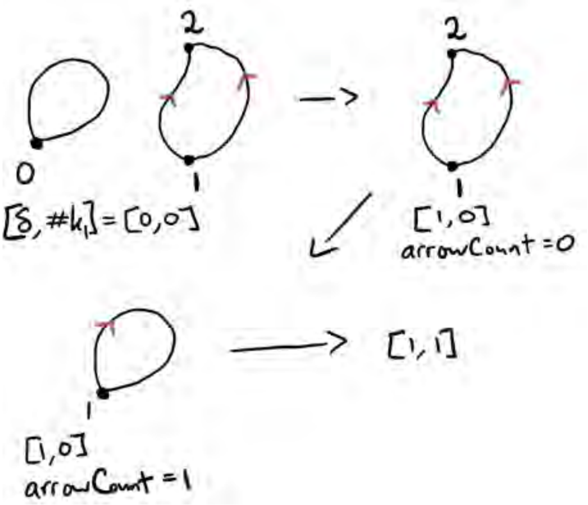
# Evaluating States

We count the number of circles and the powers of the $k_i$ in a state:

1. First, we count the number of circles that consist of only one arc. They have no $k_i$.
2. Choose an arc and search the remaining arcs for an adjacent arc. Combine them into a long arc and repeat this reduction process until a circle is made. Save the number of arrows(arcs) in the circle and increment the count of circles. (Note that every arc in this circle has an arrow, otherwise it would have been reduced. So there is no need to check for arrows.)
3. Repeat step 2 until all arcs have been counted.

We store the results in a list, where the first entry is the number of loops and the following entries are the exponents of the $k_i$.

# Example



# Pseudocode I

```
function DETERMINELOOPSANDSTATES(state)
    result ← []
    i ← 0
    while i < |state| do
        if state[i].labelOne = state[i].labelTwo then
            result[0] ← result[0] + 1
            state.pop(i)
        else i ← i + 1
        end if
    end while
    while 0 < |state| do
        first ← state[0].labelOne
        last ← state[0].labelTwo
        arrowCount ← 1
        state.pop(0)
        while first ≠ last do
            if first = state[i].labelOne then
                first ← state[i].labelTwo
                state.pop(i)
                arrowCount ← arrowCount + 1
            else if first = state[i].labelTwo then
                first ← state[i].labelOne
                state.pop(i)
                arrowCount ← arrowCount + 1
            else if last = state[i].labelOne then
                last ← state[i].labelTwo
                state.pop(i)
```

# Pseudocode II

```
            arrowCount ← arrowCount + 1
        else if last = state[i].labelTwo then
            last ← state[i].labelOne
            state.pop(i)
            arrowCount ← arrowCount + 1
        else
            i ← i + 1
        end if
    end while
    result[0] ← result[0] + 1
    if 1 ≤ arrowCount/2 then
        result[arrowCount/2] ← result[arrowCount/2] + 1
    end if
  end whilereturn result
end function
```

# Summary



The final step is to compute

$$\langle K \rangle_A = \sum_{s \in S} A^{\alpha(s)-\beta(s)}(-A^2 - A^{-2})^{\delta(s)-1}\langle s \rangle.$$

$A^{\alpha(s)-\beta(s)}$ is the weight we have associated with state $s$, $\delta(s)$ is the number of circles in $s$ we calculated on the previous page, and $\langle s \rangle = \prod_{c \in s} k_{i(c)}$ which we also calculated on the previous page.

# Putting It All Together

1. Convert Gauss code to signed planar diagram.
2. Calculate all states of planar diagram.
3. Reduce the crossings of each state.
4. Count the circles and number of arrows on each loop in each state.
5. Assemble the normalized arrow polynomial.

# Demonstration

**Demonstration Time!**

# Further Directions

- Modify the algorithm to calculate the arrow polynomial of virtual links. The program could take in a list of Gauss codes for the link components. Implementing this would not require changing any of the algorithm after states have been calculated.

# Further Directions

- Modify the algorithm to calculate the arrow polynomial of virtual links. The program could take in a list of Gauss codes for the link components. Implementing this would not require changing any of the algorithm after states have been calculated.
- Modify the program to enable the user to make substitutions. The initial motivation for this program was to help find specializations of the arrow polynomial that lead to the two Jones-type polynomials in (Boninger, 2022). This modification would be useful for exploring that direction.

# References




- Bhandari, K. (2009). Computing the Arrow Polynomial. *Rose-Hulman Undergraduate Mathematics Journal, 10*(1). https://scholar.rose-hulman.edu/rhumj/vol10/iss1/2
- Boninger, J. (2022). The Jones Polynomial from a Goeritz Matrix. *Bulletin of the London Mathematical Society, 55*(2), 732-755. https://doi.org/10.1112/blms.12753
- Chmutov, S. (2023). *The Jones, HOMFLYPT, and Arrow Polynomials*[Lecture Notes]. https://people.math.osu.edu/chmutov.1/wor-gr-su23/jones-HOMFLY-arrow%202023.pdf