Patrick Walker

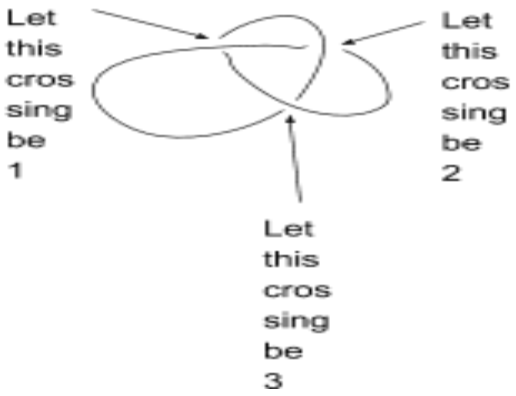
3 April 2020

Topology Project #2: Coding the Jones Polynomial

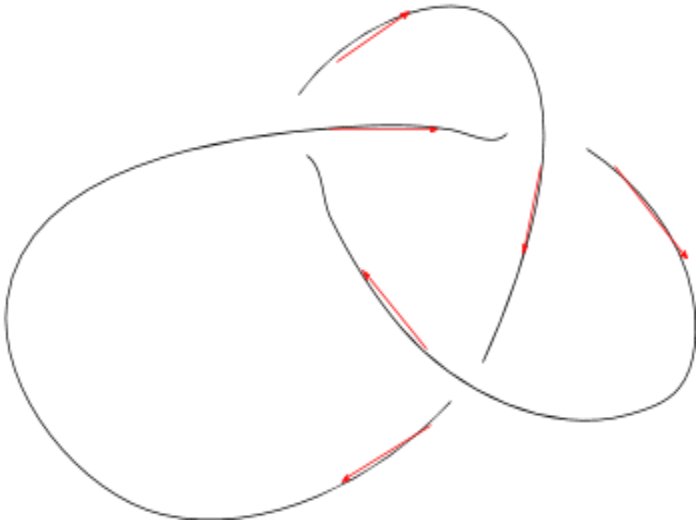
On Homework #4, our group spent 6 hours straight on one problem, the Jones Polynomial problem. The Jones Polynomial problem was such a thorn in our sides that we decided that our project should be a way to calculate the Jones Polynomials of knots quickly and accurately. Thus, we turned to coding and created a program that takes in certain parameters and, for the most part, outputs the Jones Polynomial for us. Our code takes a special parameterization of a knot that we call the Walker Parameterization (named after myself) and outputs the knot's Jones Polynomial.

The first topic of discussion is the Walker Parameterization, which can be formed through a number of steps. We will also show an example with the parameterization of the right-handed trefoil

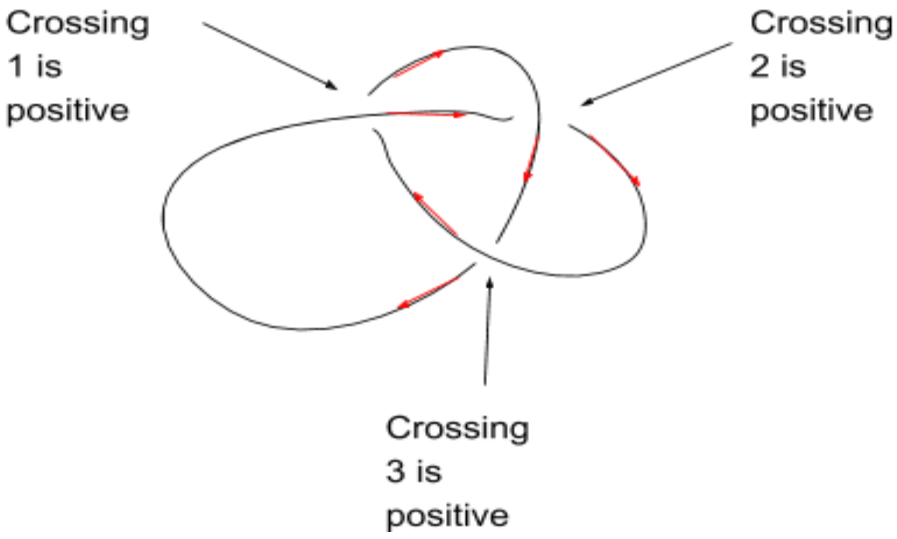
- First: assign identities to each intersection



- Third, choose a direction to go around the knot (a.k.a. An orientation)



- Second: determine whether each intersection has a positive orientation or negative orientation



- Fourth: For each complete strand you come to along the length of the knot perform the following:

- Record the intersection the strand starts at. if it is a positive orientation record a 1 with the identity of the intersection & if it is a negative orientation, record a -1.
- Record each intersection the strand passes through (overcrossing). the strand has a directionality of 0 for these intersections.
- Record the intersections and their directionality number (in this case 0) as pairs.
- Record the intersection the strand terminates at. if it is a positive orientation record a -1 for the directionality number, and if it is a negative orientation, record a 1.
- Record the intersection and the directionality number as a pair. The notation for each strand is the collection of pairs generated in this manner with the intersections in order along length of strand.

- Fifth: the notation for the knot diagram is the list of strands

Overall, the Walker parametrization for a knot diagram will look like a list of strands which are lists of pairs of an intersection and a directionality number.

```
input the parametrized knot starting on the 9th line, each line is one strand as in the example:
##########################
Example: right hand trefoil
[1,1],[2,0],[3,-1]
[2,1],[3,0],[1,-1]
[3,1],[1,0],[2,-1]
##########################
start here:
```

Steps 4 and 5 go together so we will just show the final parameterization.

This is useful for our program, for it allowed us to create a method of expressing our knots without having to physically draw out the knot and have a computer program recognize the knot based on a picture, which is extremely difficult to execute.

So once we calculate the Walker Parameterization, we can move onto the actual program. Our code consists of two parts: the functions and the ordering.

```
def complete_polynomial_bracket ():
    knot_unperturbed=collect_parametrized_knot()
    bracket=copy.deepcopy(knot_unperturbed)
    bracket=bracket_polynomial_no_removal(bracket)
    bracket=remove_knots(bracket)
    bracket=compile_coefficients(bracket)
    bracket=X(knot_unperturbed,bracket)
    bracket=convert_to_jones(bracket)
    return bracket

jones_polynomial=complete_polynomial_bracket()
print(jones_polynomial)
```

This part of our code is the ordering and it takes the various functions in our program has and combines them in one single cohesive function that takes our parameterization to our Jones Polynomial. From here, we will explain step by step the process of how the various functions work in chronological order.

```
import copy

def collect_parametrized_knot():
    information=open("knot parametrization input.txt","r")
    information=information.readlines()
    information=information[8:]
    print(information)
    information=str(information)
    information=information[2:(len(information)-2)]
    information=information.split()
    print(information)
    for i in range(len(information)):
        strand_considered=information[i]
        if i==0:
            strand_considered=strand_considered[0:len(strand_considered)-4]
        elif i==(len(information)-1):
            strand_considered=strand_considered[1:len(strand_considered)]
        else:
            strand_considered=strand_considered[1:len(strand_considered)-4]
        information[i]=strand_considered
    parametrization=list()
    print(information)
    for i in range(len(information)):
        strand_considered=information[i]
        List=strand_considered.split(',')
        for i in range(len(List)):
            item_considered=List[i]
            if i%2==0:
                item_considered=item_considered[1:]
            else:
                item_considered=item_considered[0:(len(item_considered)-1)]
            List[i]=item_considered
        new_List=list()
        for i in range(0,len(List),2):
            print(List[i+1])
            List[i+1]=int(float(List[i+1]))
            new_item=list([List[i],List[i+1]])
            new_List.append(new_item)
        List=new_List
        parametrization.append(List)
    return (parametrization)
```

This is the beginning chunk of the code and what it does is take in the Walker parameterization and converts it into a readable object for the rest of the code. From there the code takes the knot and moves on to the knot splitting portion.

```
def check_intersections(knot_unperturbed):
    intersections=list()
    a=0
    while a < len(knot_unperturbed):
        strand_considered=knot_unperturbed[a]
        b=0
        while b < len(strand_considered):
            intersection_considered=strand_considered[b]
            if intersection_considered[0] not in intersections:
                intersections.append(intersection_considered[0])
            b=b+1
        a=a+1
    return intersections
```

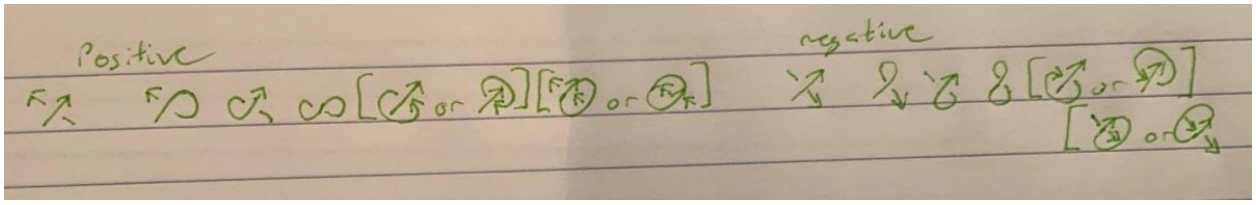
This block of code takes the identity of all intersections present in the knot and puts them in a list (This is important later in the code, for when we split the intersections, and erase the corresponding identities until there are no more entries in that list, which in turn corresponds to the point where we will have nothing left except trivial knots).

```
def reverse_strands(knot_unperturbed,initial_strand,intersection):
    knot2=copy.copy(knot_unperturbed)
    strands_to_flip=list()
    strands_to_flip.append(initial_strand)
    intersection_check=initial_strand[-1]
    a=0
    while a < len(knot2) and intersection_check[0] != intersection:
        strand_considered=knot2[a]
        intersection_considered=strand_considered[0]
        if intersection_considered[0]==intersection_check[0] and intersection_considered[1]==-(intersection_check[1]):
            strands_to_flip.append(strand_considered)
            intersection_check=strand_considered[-1]
            a=0
        else:
            a=a+1
    strands_flipped=list()
    a=0
    while a < len(strands_to_flip):
        strand_considered=strands_to_flip[a]
        knot2.remove(strand_considered)
        b=1
        strand_flipped=list()
        while b<(len(strand_considered)+1):
            strand_flipped.append(strand_considered[-b])
            b=b+1
        strands_flipped.append(strand_flipped)
        knot2.append(strand_flipped)
        a=a+1
    intersections_flipped=list()
    a=0
    while a < len(strands_flipped):
        strand_considered=strands_flipped[a]
        b=0
        while b < len(strand_considered):
            intersection_considered=strand_considered[b]
            if intersection_considered[1]==0 and intersection_considered[0] != intersection:
                intersections_flipped.append(intersection_considered[0])
            b=b+1
        a=a+1
    a=0
```

```
    while a < len(knot2):
        strand_considered=knot2[a]
        b=0
        while b < len(strand_considered):
            intersection_considered=strand_considered[b]
            if intersection_considered[0] in intersections_flipped:
                if intersection_considered[1]==1:
                    intersection_considered[1]=-1
                elif intersection_considered[1]==-1:
                    intersection_considered[1]=1
            b=b+1
        a=a+1
    a=0
    while a < len(strands_flipped):
        strand_considered=strands_flipped[a]
        b=0
        while b < len(strand_considered):
            intersection_considered=strand_considered[b]
            if intersection_considered[0] in intersections_flipped:
                if intersection_considered[1]==1:
                    intersection_considered[1]=-1
                elif intersection_considered[1]==-1:
                    intersection_considered[1]=1
            b=b+1
        a=a+1
    return knot2, strands_flipped
```

This function takes a starting strand from an intersection, and then follows the knot, collecting the strands that its path follows until it gets back to the intersection selected. Then it reverses the

direction of these strands, and replaces the original strands with these new flipped strands. However, there is more to do. Flipping the direction of an overcrossing changes an intersection's orientation from positive to negative, so this must be accounted for. In the case of the code, we collect the identity of all of the intersections whose overcrossings are in the flipped intersections. Then we look for the undercrossing portions of the intersections and reverse their directionality number (the second number in the intersection pairs). For example, consider a strand terminating at a positive orientation intersection in the original knot, its directionality number is -1. However, on flipping the overcrossing, it becomes a negative orientation, so its directionality number becomes a 1.



The above is a picture of all possibilities of intersections (when focused on an intersection itself and not the surrounding knot). As you can see there are 12 distinct possibilities of intersections. It is for this reason that the code for the actual splitting of the knot is horrendously long.
When viewing the kauffman bracket as a person, and when performing the kauffman bracket breakdowns on paper these different possibilities don't seem different when decomposing the knot, the peculiarities of the computer make it significantly more complex. Furthermore, in the case of negative orientation intersections in the A^1 bracket, you have to reverse some strands to ensure the parametrization functions properly, while performing the A^-1 bracket on positive orientation intersections requires reversing some strands as well.

```
def bracket_polynomial_no_removal(knot_unperturbed):
    bracket=list(['+',0,knot_unperturbed])
    intersections=check_intersections(knot_unperturbed)
    length_intersections=len(intersections)
    while length_intersections>0:
        intersection=intersections[0]
        new_bracket=list()
        for i in range(2,len(bracket),3):
            exponent=bracket[i-1]
            knot=bracket[i]
            knot_A=split_intersection_A(knot,intersection)
            knot_A_inverse=split_intersection_A_inverse(knot,intersection)
            new_bracket.append('+')
            new_bracket.append(exponent+1)
            new_bracket.append(knot_A)
            new_bracket.append('+')
            new_bracket.append(exponent-1)
            new_bracket.append(knot_A_inverse)
        bracket=new_bracket
        intersections.remove(intersection)
        length_intersections=len(intersections)
    bracket=compile_bracket(bracket)
    return bracket
```

```
def remove_knots(bracket_unperturbed):
    bracket=copy.deepcopy(bracket_unperturbed)
    a=0
    while a < len(bracket):
        bracket_considered=bracket[a]
        knot_considered=copy.deepcopy(bracket_considered[2])
        if len(knot_considered)> 1:
            bracket.remove(bracket_considered)
            delete=knot_considered.pop(0)
            if bracket_considered[0]=='+':
                sign='-'
            else:
                sign='+'
            exponent=bracket_considered[1]
            exponent1=exponent+2
            exponent2=exponent-2
            new_bracket_1=[sign,exponent1,knot_considered]
            new_bracket_2=[sign,exponent2,knot_considered]
            bracket.insert(a,new_bracket_1)
            bracket.insert(a+1,new_bracket_2)
        else:
            a=a+1
    return bracket
```

It then takes the knot through our set of “Kauffman Bracket functions”. The first function above is method of finding the Kauffman Bracket up to the point where we must remove trivial unknots with what we defined in class as a “C-move” and the second function above performs the said

C-moves and applies the corresponding cost by multiplying by ($-A^2-A^{-2}$) for each trivial knot removed.

```
def compile_coefficients(bracket):
    polynomial=list()
    a=0
    while a < len(bracket):
        bracket_considered=bracket[a]
        holder1=bracket_considered[0]
        holder2=bracket_considered[1]
        b=0
        added=False
        while b<len(polynomial) and added==False:
            collected_considered=polynomial[b]
            if holder2 == collected_considered[1]:
                if holder1=='+':
                    collected_considered[0]=collected_considered[0]+1
                elif holder1=='-':
                    collected_considered[0]=collected_considered[0]-1
                added=True
                b=b+1
            else:
                b=b+1
        if b==len(polynomial) and added==False:
            holder1=bracket_considered[0]
            holder2=bracket_considered[1]
            if holder1=='+':
                holder3=1
            elif holder1=='-':
                holder3=-1
            holder4=list([holder3,holder2])
            polynomial.append(holder4)
        a=a+1
    a=0
    n = len(polynomial)
    while a < n:
        exponent_considered=polynomial[a]
        if exponent_considered[0] == 0:
            polynomial.remove(exponent_considered)
            n=len(polynomial)
        else:
            a=a+1
    return polynomial
```

Then, this part of the code compiles all of the information together and determines how many multiples of each exponent are present, or in other words, this part of the code takes all of the calculations the other functions performed and outputs the Alexander Polynomial result found through the collection of the previous parts.

```
def writhe_number(knot_unperturbed):
    w=0
    intersections=check_intersections(knot_unperturbed)
    for i in range(len(intersections)):
        a=0
        while a < len(knot_unperturbed):
            strand_considered=knot_unperturbed[a]
            intersection_considered=strand_considered[-1]
            if intersection_considered[0]==intersections[i]:
                if intersection_considered[1]==-1:
                    w=w+1
                if intersection_considered[1]==1:
                    w=w-1
            a=a+1
    return w
```

```
def X (knot_unperturbed,bracket):
    knot=copy.deepcopy(knot_unperturbed)
    writhe=writhe_number(knot)
    writhe=-3*writhe
    for i in range(len(bracket)):
        bracket_considered=bracket[i]
        bracket_considered[0]=bracket_considered[0]*-1
        bracket_considered[1]=bracket_considered[1]+writhe
    return bracket
```

Then, our code moves on to the calculation of the correction factor in two parts. The first piece of code finds the writhe number of the original knot and then sends it to the second piece of code that takes it, calculates the correction factor, and then applies it to our Alexander Polynomial from before.

```
def convert_to_jones(bracket_unperturbed):
    bracket=copy.deepcopy(bracket_unperturbed)
    for i in range(len(bracket)):
        bracket_considered=bracket[i]
        bracket_considered[1]=-bracket_considered[1]/4
    return bracket
```

And then, for the final step, this piece of the code takes our Alexander Polynomial and converts it into our Jones Polynomial by converting A to t through the relation $A=t^{1/4}$ thus concluding the explanation of our code.