Explorations on Conjugations of Local Rotations

by

Katie Kruzan

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Masters of Science in Industrial Mathematics

Middle Tennessee State University
November 2021

Thesis Committee:

Dr. Chris Stephens, Chair

Dr. James Hart

Dr. Xiaoya Zha

Dr. Dong Ye

ABSTRACT

The process of manipulating single local rotations of vertices can represent standard topological graph theory techniques to add vertices or edges to an existing embedding. This paper looks to find patterns of these local rotations we can generalize. We start by reviewing the local rotations of vertices of small degree (3, 4, and 5). At those local rotations, we propose some hypotheses to be tested on local rotations of higher degree. We then move to outline an algorithm that can be used to analyze vertices of larger degrees programmatically.

DEDICATION

I dedicate this thesis to my mother and friends who helped me push through these last few semesters. They have given me support and continued to challenge and push me when I did not care to myself.

ACKNOWLEDGEMENTS

I would like to thank Dr. James Hart, Dr. Xiaoya Zha, and Dr. Dong Ye for taking the time to read this thesis. I also want to thank them for setting aside the time for my defense. Each of them helped me along the way for my MTSU experience, whether it be through a class or through helping getting paperwork in order. Their aid greatly helped improve my experience.

I would also like to thank Dr. Chris Stephens for spending time with me over the course of the past few semesters working through these problems. This process and his encouragement made the process of contributing to the sum of mathematical knowledge a greatly positive and exciting thing. In our busy lives, it was nice to have a reason to sit and stare at a whiteboard for a few hours at a time.

I would also like to thank my academic advisor Dr. Zachariah Sinkala. He worked with me many summers to explore topics I would not have had the chance to otherwise, expanding my joy and appreciation towards mathematics. He also was integral to helping me navigate and plan out my semesters in a way that allowed me to be presenting this thesis this semester.

Contents

List of Figures

Chapter 1

Introduction

In this problem, we are looking to determine how rotations or vertex orderings at a single vertex affect the topology of an embedding. Let G be a graph with n vertices. We define a *local rotation* at vertex v of G as a clockwise ordering of the oriented edges originating at v. A *rotation system* of the graph G consists of a set of n rotations, one for each vertex of G [1]. Work has been done analyzing the planarity of graphs with rotation systems [2], but not much work has been done on analyzing general patterns when looking at a single vertex. Motivation comes from two small, standard surgeries used in topological graph theory to add vertices or edges to an existing embedding. These surgeries are topological, whereas graph embeddings are often viewed from a purely combinatorial standpoint. Thus, one may attempt to describe the surgeries combinatorially: in effect, they are particular alterations of a single local rotation of an embedding graph, easily described in terms of conjugating by permutations.

Let 0 be a vertex in an embedding μ that is adjacent to vertices 1, 2, 3, and 4. Let's also say we have the edges 12, 23, 34, and 41. Let the triangular faces 012, 023, 034, and 041 appear in μ consecutively, in this order about 0. If one wishes to, say, add the edge 04 to the graph without changing the Euler genus by more than one, she might accomplish this by "merging" the triangles 012 and 034 with a crosscap which

intersects the graph only in the edges 02 and 03. When one does so, one needs to "switch" the edges 02 and 03 in the local rotation at 0 for the rest of the embedding to remain the same as before. This face merging procedure has proven fruitful in other studies [3] with great success to obtain minimal nonorientable embeddings of $K_{m+1,m,n}$ from minimal nonorientable embeddings of $K_{m,m,n}$. This procedure has also been used [4] to convert non-orientable surfaces to orientable surfaces.

However, this is a topological procedure, and the question then becomes, can we look at this process through a combinatorial lens? The answer is yes. The local rotation at 0 is a permutation of the edges incident with 0; for simplicity, since all such edges have 0 as an endpoint, let us drop the 0's and refer to the edge 01, for example, as just 1. In this case the procedure above changes the local rotation at 0 from (...1234...) to (...1324...), and changes the signatures on the edges 02 and 03. In other words, the change is accomplished by conjugating the local rotation at 0 by the involution (23), and by changing the edge signatures on 02 and 03.

If one prefers to avoid changes in edge signature, there is a similar procedure involving only orientable embeddings. Suppose our embedding contains consecutive triangles 012, 023, 034, 045, and 056 in this order about 0. We may merge 012, 034, and 056 by doing the following: we glue the two ends of a handle to the interiors of 012 and 056, then "re-route" the edges 02 and 03 across the handle. Topologically, one checks that the triangles 023 and 045 are still intact, but the other three triangles have been destroyed and replaced by one large face. Combinatorially, the new embedding is exactly the old embedding with the local rotation at 0 replaced by the local rotation at 0 conjugated by the permutation (24)(35). (Again, This face merging procedure has been used with great success to obtain minimal orientable embeddings of $K_{m+1,m,n}$, $K_{m+2,m,n}$, and $K_{m+3,m,n}$ from minimal orientable embeddings of $K_{m,m,n}$ in some of our unpublished results on the orientable genus of $K_{l,m,n}$).

We now look to determine patterns that can be generalized for conjugations of all types. Specifically, we will be looking for patterns of the effects these local conjugations or local rotations might have on the face topology. Consider the graph and embedding μ mentioned before. If we "switched" the edges 02 and 03 to get a new embedding γ, we would then say the local rotation to get from μ to γ is (23). That is, the edge originally ending at vertex 2 now ends at vertex 3 and vice versa.

Considering the faces, the μ embedding has four faces. 012, 023, 034, and 041. The new γ embedding now has two faces: The original 041 face and then a new face 0120230120 when walking on the right side of the edges. For simplicity, denote the outer edges of the μ faces 012, 023, 034, and 041 as I, II, III, and IV respectively. So edge 12 will be denoted I, and similarly for the rest. Then, omitting the writing of the edges back to the origin, we can denote the faces of the γ embedding as the IV face, and a new face I III II.

Consider a new local rotation (24) from μ to an embedding ϕ. That is, the edge originally ending at vertex 2 now ends at vertex 4 and vice versa. When looking at the faces, the embedding ϕ has two faces: the face I III and the face II IV. A different local rotation gives a notably different structure in the final face topology of the graph.

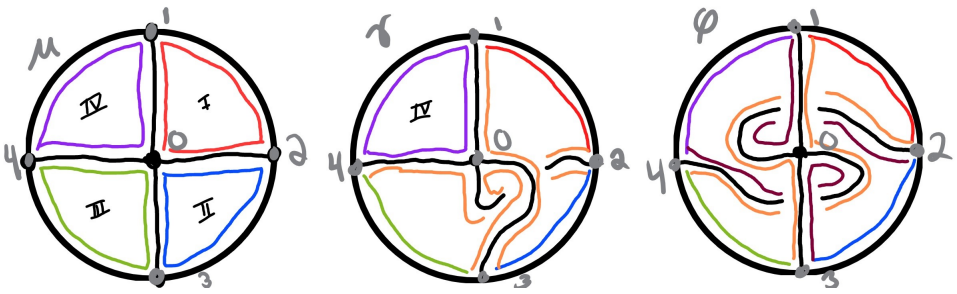

Figure 1.1: Examples of local rotations

The question then becomes, how do these local rotations affect face topology, and furthermore, can we determine any patterns to begin to make generalizations? This

paper will expound on some initial findings in local rotations of small order, and then introduce an algorithm that can be utilized to find patterns on local rotations of larger orders.

Chapter 2

Methods

2.1 Ground Rules

To keep things interesting, we only wanted to analyze local rotations that are structurally unique. For this reason, we decided to go by the following ground rules.

First off, we kept the edge 01 stationary for all rotations we explored. Consider local rotations on 3 vertices. The rotation (23) and (12) are structurally similar as you could rotate one about the origin to get the other. Similarly, all possible rotations are structurally similar to a rotation of the same order with the edge 01 stationary.

Secondly, we assume the outside edges are singular edges. These findings can easily be extrapolated to graphs with the outside edges being walks, especially since we're primarily concerned with face topology. For example, consider graph G with the vertices $0, 1, 2$, and 3 with the edges $01, 02, 03, 12, 23$, and 31. Graph G is structurally similar with regards to faces to graph F with vertices $0, 1, 2, 3$ and a with the edges $01, 02, 03, 12, 23, 3a$, and $a1$.

We also know that by Euler's formula, the number of faces can only decrease by a multiple of two since we're drawing in the plane. By this, we can determine the possible number of resulting faces when looking at these graphs. If we're looking

at a graph with an even number of faces, the smallest number of faces after a local rotation is 2. Similarly, the smallest possible number of faces for a graph with an odd number of faces after a local rotation is 1.

2.2 The Early Days

Like all good mathematicians, we started small to begin to look for patterns.

2.2.1 3 vertices

To begin, we drew all possible unique local rotations at 3 vertices. Keeping edge 01 stationary, the only local rotation we can perform is (23). The starting graph and the conjugated graph are pictured in Figure 2.1.

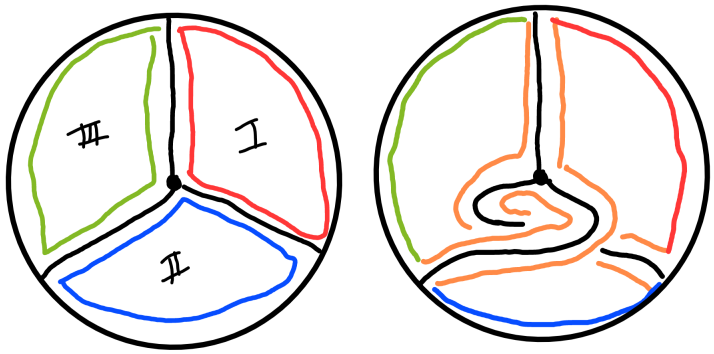

Figure 2.1: Conjugations on 3 vertices

This shows there is only 1 structurally unique local rotation (23). That rotation results in a single face: I III II.

2.2.2 4 vertices

Considering local rotations with 4 vertices, combinatorics will show the list of all possible local rotations to be (23), (24), (34), (234), and (243). The resulting 3 unique structures are shown in Figure 2.2.

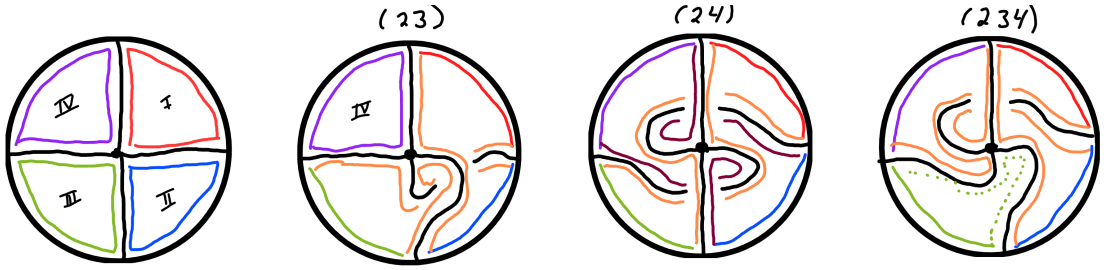

Figure 2.2: Conjugations on 4 vertices

The first structure can be generated with rotations (23) and (34). This results in 2 faces. The rotation (23) gives faces IV and I III II. The rotation (34) gives a similar structure that can be found by rotating the graph resulting from the local rotation (23) about the origin and relabeling. The second structure of local rotations with 4 vertices can be generated with the rotation (24). This results in 2 faces: I III and II IV. The third structure can be generated with rotations (234) and (243). The rotation (234) gives the 2 faces I IV II and III. This structure is similar to the (23) structure but is shifted so that face III has neither of its original edges. The rotation (243) looks like the (234) rotation flipped along the vertical axis.

From these rotations we begin to see possible patterns. Firstly, say we have a rotation (ABCDE). Based on what we see from the 4 vertices, we might propose the (AEDCB) local rotation will result in a similar structure to (ABCDE) but flipped along the vertical axis. We kept this in mind as we looked at local rotations of higher order.

Another pattern that emerged is unmoved faces can be condensed into single edges when analyzing structure. Consider the rotation of (23) on 4 vertices. This structure seems very similar to if we treated the unaffected face (IV in the (23) rotation) as a single edge in a (23) rotation with 3 vertices. This begins to show the fact that it does not matter how many faces we have together. If they stay unmoved, they are structurally similar to a single edge in rotations of a smaller order. For example, if we

looked at local rotations with 53 vertices, yet only rotated two adjacent vertices, it would be structurally similar to a (23) rotation on 3 vertices. So from here onwards, we will only review structures that do not have adjacent non-changing edges that can be 'condensed' to a single non changing edge to emulate a rotation of smaller order.

2.3 Making the Computers Work For Us

As we looked to draw local rotations of higher-order, we began to realize some patterns when it came to what faces resulted from a local rotation. The drawing and tracing faces became very systematic, and we began to determine an algorithm we could derive that would allow us to get a computer to 'draw' the local rotations and the resulting faces.

2.3.1 New Notation

To better achieve this goal, we need to differentiate between the sides of the edges that connect to the origin as opposed to the outer edges. To better do this, we will create a new structure to analyze. To do this, we will have 'inner nodes' that connect directly to the origin, and 'outer nodes' that connect to both these inner nodes and the external edges. For example, consider graph G with vertices $0, 1, 2, 3, a, b$, and c and edges $0a, 0b, 0c, a1, b2, c3, 12, 23$, and 31. This graph would be a new representation of a graph with 3 external vertices as we have talked about before. With this new notation, a (23) local rotation on 3 vertices would be removing the edges $b2$ and $c3$ and replacing them with new edges $b3$ and $c2$ respectively. An example of this process is found in Figure 2.3

In general, we will be denoting the inner nodes with lowercase letters, the outer nodes with natural numbers, and the outer segments with roman numerals. Walking

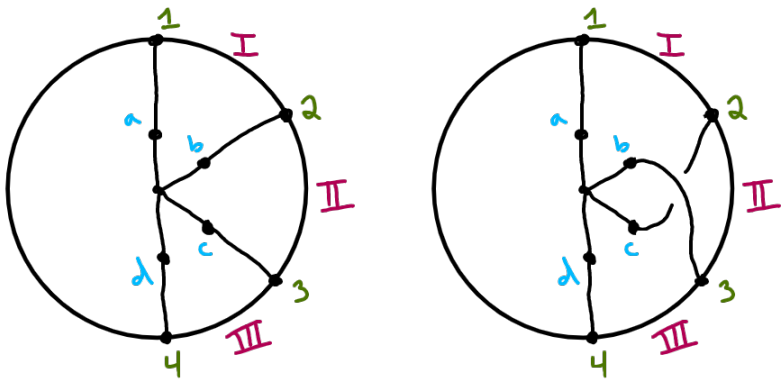

Figure 2.3: An example of before and after a (23) local rotation with the new notation

along the right side of the edges going clockwise, you can represent the starting faces for 3 vertices as a1I2ba, b2II3cb, and c3III1ac. You can represent the faces after the local rotation (23) as a single face a1I2cb3III1ac2II3ba. This can be algorithmically built.

2.3.2 The Proposed Algorithm

To walk the face a particular inner node is in, we can propose the following algorithm.

1. We're going to keep a list as the face.
2. Add the starting inner node to the face
3. Move to the adjacent outer node
4. Move to the segment to the right of the outer node
5. Move to the outer node to the right of the segment
6. Move to the adjacent inner node of the last outer node
7. Move to the inner node to the left of the current inner node.
8. If the current node does not equal the starting node, then repeat steps 3-8.

2.3.3 Walking the Edges - An Example

We first consider what the rotation of (23) is doing in this new notation. This is saying whenever we get to vertex b, we then next go to vertex 3 and vice versa. Also whenever we get to vertex c, we go to vertex 2 and vice versa. So combining our starting faces with this new information, we can begin to construct the resulting faces. We start with vertex a out of convention. Looking at our initial faces, we have a1I2 as the start to our final face. However, we no longer have the edge from 2 to b. Instead, we have an edge from 2 to c. That makes our final face have the start of a1I2c.

That brings us back to the origin. Now we're faced with the question of what inner vertex is next? Our two options are the two adjacent vertices of b and a. Since we're coming in on the right side of the vertex c, we will need to go to the vertex to the right of vertex c, which is vertex b. So that makes the start of our final face to be a1I2cb so far. We now know that the b vertex goes to the 3 vertex, giving a1I2cb3.

Every time we get to an outer vertex, the outer segment is unchanged from the original. Since we're only messing with the edges between the outer and inner nodes, we are guaranteed to know the next outer segment and outer node whenever we get to an outer node from an inner node. Because of this, we know the start of the final face is a1I2cb3III1. Using similar logic as described above, you continue to build the face until you start repeating. This results in the face a1I2cb3III1ac2II3ba.

So we know this notation and algorithm works for this example. The question then becomes "Can we algorithmically generate the list of faces for all graphs of this form?"

2.3.4 Proving the Proposal

Before we start to generate this programmatically for all graphs of this form, we need to prove the logic holds and can be extrapolated to graphs of all orders. By convention, we will always walk on the right side of an edge. This means that every time we're going in and out of the origin, we will be coming in on the right side of an inner vertex, which will be followed by leaving on the right side of the inner vertex directly before it in the labeling we described above.

We also know that since we are not messing with the relation between the outer vertices and the outer segments, if we are staying on the right side of the edges, we are guaranteed the segments will have the same outer vertices as before. This means every time we approach vertex 1 on the right side, we will then go to the segment I followed by outer vertex 2. This is similar for all the other outer vertices and segments.

These two facts together, mean our algorithm holds with our new notation for the general case. We have spent time creating a python program that implements the algorithm describes. The code is in the appendix.

2.3.5 The Implementation of the Algorithm

The initial implementation of this algorithm involves Python code, as that is the primary code of choice for the author. To implement the code, we utilized a concept core to Python's oriented nature: Python classes. This gave us the ability to more seamlessly create objects that are able to individually keep track of what they are adjacent to. We created three types of objects: inner nodes, outer nodes, and outer segments.

Inner nodes are the nodes that connect the outer nodes to the origin. Outer nodes are the nodes that connect the inner nodes to the outside edges or segments.

Segments are the outer edges connecting outer nodes to one another.

Inner node objects keep track of three elements: the outer node they are adjacent to and the inner nodes to the left and right with respect to the origin. Outer node objects keep track of the inner node they are adjacent to and the outer segments to the left and right with respect to the origin. Lastly, segment objects keep track of the two outer nodes they are adjacent to on the left and right. Each object has methods to get the current related objects and to set them as well. For ease of programming implementation and readability, we give outer nodes labels of the natural numbers $(1, 2, 3, \dots)$, inner nodes labels of the lowercase alphabet $(a, b, c, \dots)$, and outer segments with labels of negative natural numbers $(-1, -2, -3, \dots)$.

The program itself is designed to take an input of the number of vertices we want to analyze and the rotation we want to perform. The program then creates the starting vertex set up based on the number of nodes. The starting circle is defined by a list of the segment, outer node, and inner node objects. It then loops through all the inner nodes and generates the faces they are each a part of. We then write these faces in a canonical ordering (starting from the lowest segment label value) to make it easy to compare duplicate faces. After these faces are in a canonical ordering, we get rid of duplicate faces to aid in readability and ease of analysis.

Once the first circle is created and the starting faces are determined, the program then starts to perform the local rotations. It does this by switching the edges between the outer and inner nodes based on the rotation given.

Then we go through the process of listing the distinct faces again. And this results in the faces after the local rotation.

This computer program allows us to generate the graphs resulting local rotations in a matter of seconds rather than tediously drawing and tracing ourselves on paper. This program can be expanded to systematically generate these rotations and begin

to perform simple analyses on them. Questions we could answer with the addition of a few lines of code include "How many structurally distinct local rotations are there of order 10?" We decided to utilize this program to help us analyze our proposed hypotheses at scale.

Chapter 3

Results

This notation and algorithm give us the power to analyze local rotations of higher order with ease. It also provides us with a new tool to both describe and analyze these resulting faces. In particular, we can begin to test the theories we began to hypothesize in smaller ordered rotations. For example, we can test our theory that a rotation of any size (ABCD...Z) is structurally similar to the rotation (Z...DCBA) but flipped along the vertical axis. To begin to understand the power of these, let's look at the cases of 5 vertices and some local rotations of 9 vertices.

3.1 5 Vertices

When thinking about the structures that have adjacent non-changing edges, the local rotations of (23) $\cong$ (34) $\cong$ (45), (24) $\cong$ (35) $\cong$ (25), and (234) $\cong$ (345) $\cong$ (243) $\cong$ (354) are all structurally similar to local rotations of smaller order. So we will analyze other rotations. Simple combinatorics shows the list of all possible local rotations on 5 vertices excluding those above is as follows.

- (235)
- (245)
- (253)
- (254)
- (2345)

- (2354)
- (2453)
- (2543)
- (24)(35)
- (2435)
- (2534)
- (23)(45)
- (25)(34)

Drawings of each of these can be found in Figure 3.1

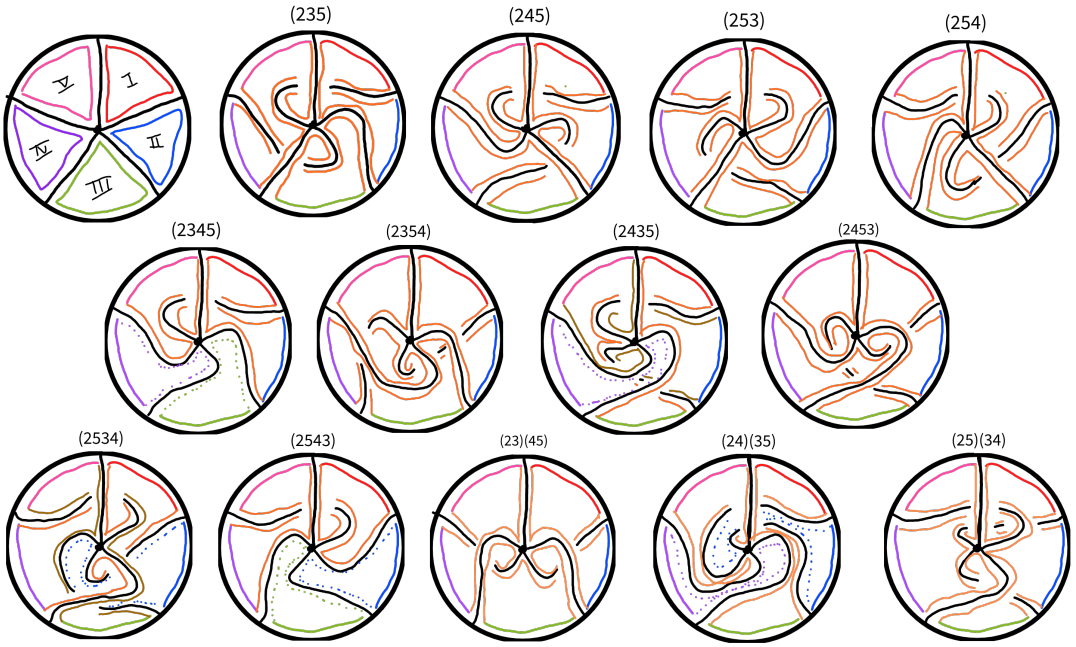

Figure 3.1: Conjugations on 5 vertices

With thirteen local rotations, and many of them creating single faces, we have a need to formally describe the structure of the resulting graphs after undergoing local rotations. We can continue to label the rotations in a way that will give the order for which the outside edges appear. We can also describe the resulting graphs in terms of the number of edge crossings required to draw the graph on the plane. For example, the rotation (235) results in a single face I IV III V II and it must be drawn with two edge crossings.

Let's consider the local rotations of (2345) and (2543). Based on our previous hypothesis we generated when looking at rotations of order 4, these two seem like prime candidates for being structural mirrors of each other. When looking at the two drawings, they look like exact mirror flips of each other. To confirm this, let's define

a function σ that takes an input and outputs the corresponding element reflected along the vertical axis. For example, in a starting local rotation of order 5, σ of face IV would be face II. With this notation, for graph G to be a mirror of graph H, the clockwise ordering of faces of G should be equal to the counterclockwise σ(faces of H). For example, in the resulting graph after local rotation (2345) we see the faces I V II, III, and IV. If the resulting graph after the local rotation (2543) is a mirror of the graph after the (2345) rotation, the resulting faces of (2543) should be I V IV, III, and II. Upon observation of the two drawings, you can determine this is true.

For two graphs to be structural mirrors, they must be the same in every way other than the fact they are flipped along the vertical axis. So another thing we may check is the number of edge crossings that are forced when we draw it in the plane. Both (2345) and (2543) result in a single forced edge crossing. We can then confirm (2345) and (2543) are structural mirrors of each other. This is a promising observation that confirms our hypothesis. We went to then test other possible pairs of rotations.

Let's consider the local rotations of (2354) and (2453). When observing the resulting faces of these local rotations, we get that (2354) has a single face I V IV III II. The mirror flip of this should have the resulting face of I II III IV V. Looking at the resulting graph of a (2453) rotation, it has a single face I IV II V III. Looking at this comparison, these two are not to be mirrors of each other. So our hypothesis is false, with this counterexample as proof.

Instead of thinking about these mirrors in the terms of how we write their local rotations, let's think about these in terms of our new notation that includes inner and outer nodes. The local rotation (2354) is getting replacing the edges $b2, c3, d4$, and $e5$ with the edges $b3, c5, e4$, and $d2$. If we put our final edges through the previously described *sigma* function, we get $\sigma(b3) = e4$, $\sigma(c5) = d2$, $\sigma(e4) = b3$, $\sigma(d2) = c5$. We see $\sigma(b3, c5, e4, d2) = e4, d2, b3, c5 = b3, c5, e4, d2$ which is our original set of edges

in the (2354) rotation. through this lens, its easy to see that the mirror of (2354) is itself. We can also look at putting the rotation itself through the sigma function. $\sigma(2354) = (5423) = (2354)$.

We can also begin to generalize what the mirror for any given rotation will be. If we put the rotations themselves through the σ function, then we are able to get the rotations of their mirrors. This also works for local rotations that do not include all available vertices. Consider the local rotation (235) on 5 vertices. This can also be represented as (235)(4) with 4 as a singleton that does not get changed. We can then put that through the sigma function to determine the mirrored rotation. We get $\sigma((235)(4)) = (542)(3) = (254)$ and when looking at the drawings of the rotations for (235) and (254) we can see they are mirror images of each other.

3.2 9 Vertices

The ease of running the python script allows us to spot-check this theory for larger ordered local rotations with ease. We utilized the program to analyze the local rotation (29673845) and see if it produced the expected mirror. By our earlier hypothesis, the rotation (25483769) should be the structural mirror of (29673845).

The local rotation (29673845) results in 3 faces: I V IV VIII, II VII IX VI, and III. The expected mirror would result in the face rotations of σ(I V IV VIII) = IX V VI II = II IX V VI, σ(III VII IX VI) = VIII III I IV = I IV VIII III, and σ(III) = VIII. The local rotation (25483769) gives 3 faces: II VI V XI = counter-clockwise σ(I V IV VIII), I IV VIII III = counter-clockwise σ(III VII IX VI) and VII = σ(III). This is exactly what we expected. Spot checking around in the local rotations of order 9 gives similar results.

Chapter 4

Possible Improvements

There is also potential for exploration in implementing it in different languages more utilized by mathematicians, such as MATLAB or R. This would improve accessibility to mathematicians, and therefore increase the potential for further exploration. Even within the used language Python, there is room for performance improvement especially as rotations of extremely high order are explored.

There is also room to simplify this algorithm, as the segment object might be entirely unnecessary. This is because the segment connects outer vertices with each other in the same manner each time.

Chapter 5

Conclusions

Our work on this project has laid the groundwork for future exploration on this subject. We believe there are patterns to be discovered here that were not incredibly apparent at lower orders. By the nature of the proposed algorithm and implementation, it is primed for larger implementation and analysis. There is also work that could be done to analyze the results at a larger scale than the current implementation does. As we continued to explore this topic, we found ourselves thinking of possible questions we left unexplored. Can we find a local rotation that will give us one or two final faces for local rotations of all orders? As n approaches infinity, what does the distribution of the number of resulting faces tend to? Can we begin to decompose local rotations or possibly be able to build them from combinations of other local rotations? These are left as open questions for the reader to explore.

Appendix A

Python Code

A current copy of this code is posted below. This code will be kept up to date an accessible through https://github.com/katiekruzan/masters-thesis

```
"""
Here we're going to code for the local rotations. We're doing an
    object oriented approach
Left and right are in reference to the origin
"""

__version__ = 1.0
__author__ = 'Katie Kruzan'

import string  # just to get the alphabet easily iterable
import sys  # This just helps us in our printing
from typing import Dict  # This helps us in our documentation


# Getting the structure for the classes we're putting together
class Segment:
    """
```

```
17      These are going to represent the outer segments and the
    mysteries they hold.
18      The segments will be adjacent to 2 outer nodes
19      """
20
21      def __init__(self, name: str):
22          """
23          Initialize the segment, keeping a place for the right left
    outer vertices to which it is adjacent
24          :param name: How we will reference this segment. In this
    implementation, it is expected to be a negative integer
25          """
26          self.leftOuter = None
27          self.rightOuter = None
28          self.name = name
29
30      def getName(self) -> str:
31          """
32          Return the name we gave to this segment.
33          :return: name
34          """
35          return self.name
36
37      def getLeftOuter(self):
38          """
39          Return the outer node to the left of this segment with
    respect to the origin
40          :return: leftOuter
41          """
42          return self.leftOuter
43
44      def getRightOuter(self):
```

```
45          """
46          Return the outer node to the right of this segment with
    respect to the origin
47          :return: rightOuter
48          """
49          return self.rightOuter
50
51      def setLeftOuter(self, left):
52          """
53          Set the outer node to the left of this segment with respect
    to the origin
54          Also, set left's right segment to this segment.
55          :param left: A outer node object to be referenced as this
    segment's left outer node
56          :return: None
57          """
58          self.leftOuter = left
59          if left.getRightSegment() is None:
60              left.setRightSegment(self)
61
62      def setRightOuter(self, right):
63          """
64          Set the outer node to the right of this segment with respect
     to the origin
65          Also, set right's left segment to this segment.
66          :param right: A outer node object to be referenced as this
    segment's right outer node
67          :return: None
68          """
69          self.rightOuter = right
70          if right.getLeftSegment() is None:
71              right.setLeftSegment(self)
```

```
72
73      def isValidObject(self) -> bool:
74          """
75          Checks to see if this segment has been full initialized.
76          :return: valid returns true if it has both the left and
    right outer nodes set
77          """
78          if (self.leftOuter is None) or (self.rightOuter is None):
79              return False
80          return True
81
82      def toString(self) -> str:
83          """
84          Returns a formatted string of the left and right outer nodes
     this is associated with
85          :return: Description string
86          """
87          return 'left Outer: ' + self.leftOuter.getName() + '\nright
    Outer: ' + self.rightOuter.getName()
88
89
90  class Outer:
91      """
92      Class to represent the outer vertices that are adjacent to an
    inner vertex and 2 outer segments
93      """
94
95      def __init__(self, name: str):
96          """
97          Initialize the outer node
98
```

```
99          Keeping a place for the inner vertex and right and left
    outer segments to which it is adjacent.
100         :param name: How we will reference this outer node. In this
    implementation, it is expected to be a positive integer
101         """
102         self.adjInner = None
103         self.leftSegment = None
104         self.rightSegment = None
105         self.name = name
106
107     def getName(self) -> str:
108         """
109         Return the name we gave to this outer node.
110         :return: name
111         """
112         return self.name
113
114     def getLeftSegment(self) -> Segment:
115         """
116         Return the segment object to the left of this outer node
    with respect to the origin
117         :return: leftSegment
118         """
119         return self.leftSegment
120
121     def getRightSegment(self) -> Segment:
122         """
123         Return the segment object to the right of this outer node
    with respect to the origin
124         :return: rightSegment
125         """
126         return self.rightSegment
```

```
127
128     def getAdjInner(self):
129         """
130         Return the inner node object adjacent to this outer note
    object
131         :return: adjInner
132         """
133         return self.adjInner
134
135     def setLeftSegment(self, left: Segment):
136         """
137         Set the segment to the left of this outer node with respect
    to the origin
138         Also, set left's right outer node to self.
139         :param left: A segment object to be referenced as this node'
    s left outer segment
140         :return: None
141         """
142         self.leftSegment = left
143         if left.getRightOuter() is None:
144             left.setRightOuter(self)
145
146     def setRightSegment(self, right: Segment):
147         """
148         Set the segment to the right of this outer node with respect
     to the origin
149         Also, set right's left outer node to self.
150         :param right: A segment object to be referenced as this node
    's right outer segment
151         :return: None
152         """
153         self.rightSegment = right
```

```
154         if right.getLeftOuter() is None:
155             right.setLeftOuter(self)
156 
157     def setAdjInner(self, inner):
158         """
159         Set the inner node adjacent to this outer node
160         Also, set inner's adjacent outer node to self.
161         :param inner: A inner node object to be referenced as this
    node's adjacent inner node
162         :return: None
163         """
164         self.adjInner = inner
165         if inner.getAdjOuter() is None:
166             inner.setAdjOuter(self)
167 
168     def isValidObject(self) -> bool:
169         """
170         Checks to see if this outer node has been full initialized.
171         :return: valid returns true if it has the left segment,
    right segment, and inner node set
172         """
173         if (self.leftSegment is None) or (self.rightSegment is None)
     or (self.adjInner is None):
174             return False
175         return True
176 
177     def toString(self) -> str:
178         """
179         Returns a formatted string of the left segment, right
    segment, and inner node this outer node is associated with
180         :return: Description string
181         """
```

```
182         return 'left Segment: ' + self.leftSegment.getName() + '\
    nright Segment: ' + self.rightSegment.getName() \
183                 + '\nadj Inner: ' + self.adjInner.getName()
184
185
186 class Inner:
187     """
188     Class to represent the inner vertices that are adjacent to an
    outer vertex and 2 neighboring inner vertices
189     """
190
191     def __init__(self, name: str):
192         """
193         Initialize the inner node object
194
195         Keeping a place for the outer vertex and right and left
    adjacent inner nodes.
196         :param name: How we will reference this inner node. In this
    implementation, it is expected to be a lowercase letter
197         """
198         self.adjOuter = None
199         self.leftInner = None
200         self.rightInner = None
201         self.name = name
202
203     def getName(self) -> str:
204         """
205         Return the name we gave to this inner node.
206         :return: name
207         """
208         return self.name
209
```

```
210     def getLeftInner(self):
211         """
212         Return the inner node object to the left of this inner node
    with respect to the origin
213         :return: leftInner
214         """
215         return self.leftInner
216 
217     def getRightInner(self):
218         """
219         Return the inner node object to the right of this inner node
     with respect to the origin
220         :return: rightInner
221         """
222         return self.rightInner
223 
224     def getAdjOuter(self) -> Outer:
225         """
226         Return the outer node object adjacent to this inner node
227         :return: adjOuter
228         """
229         return self.adjOuter
230 
231     def setLeftInner(self, left):
232         """
233         Set the inner node to the left of this inner node with
    respect to the origin
234         Also, set left's right inner node to self.
235         :param left: An inner node object to be referenced as this
    node's left inner node
236         :return: None
237         """
```

```
238         self.leftInner = left
239         if left.getRightInner() is None:
240             left.setRightInner(self)
241 
242     def setRightInner(self, right):
243         """
244         Set the inner node to the right of this inner node with
    respect to the origin
245         Also, set right's left inner node to self.
246         :param right: An inner node object to be referenced as this
    node's right inner node
247         :return: None
248         """
249         self.rightInner = right
250         if right.getLeftInner() is None:
251             right.setLeftInner(self)
252 
253     def setAdjOuter(self, outer: Outer):
254         """
255         Set the outer node adjacent to this inner node
256         Also, set outer's adjacent inner node to self.
257         :param outer: An outer node object to be referenced as this
    node's adjacent outer node
258         :return: None
259         """
260         self.adjOuter = outer
261         if outer.getAdjInner() is None:
262             outer.setAdjInner(self)
263 
264     def isValidObject(self) -> bool:
265         """
266         Checks to see if this inner node has been full initialized.
```

```
267         :return: valid returns true if it has the left inner node,
    right inner node, and adjacent outer node set
268         """
269         if (self.leftInner is None) or (self.rightInner is None) or
    (self.adjOuter is None):
270             return False
271         return True
272 
273     def toString(self) -> str:
274         """
275         Returns a formatted string of the left inner node, right
    inner node, and adjacent outer node this inner node
276         is associated with
277         :return: Description string
278         """
279         return 'left Inner: ' + self.leftInner.getName() + '\nright
    Inner: ' + self.rightInner.getName() \
280                + '\nadj Outer: ' + self.adjOuter.getName()
281 
282 
283 def standardCircle(num_verts: int) -> (Dict[str, Segment], Dict[str,
    Outer], Dict[str, Inner]):
284     """
285     This will go through and initialize our standard starting circle
286     :param num_verts: the number of outer nodes we will have
287     :returns: tuple(segs, outs, inns)
288         -segs - dictionary of str: Segment objects in the circle \\
289         -outs - dictionary of str: Outer objects in the circle \\
290         -inns - dictionary of str: Inner objects in the circle
291     """
292     # Initializing our dictionaries
293     segs = dict()
```

```
294     outs = dict()
295     inns = dict()
296 
297     # Running through the number of vertices we will be edning up
    with
298     for i in range(num_verts):
299         # start with an inner node - labeling with lowercase letters
300         inn = Inner(string.ascii_letters[i])
301         # If we aren't on the first one, connect it to the previous
    one.
302         if i != 0:
303             inn.setLeftInner(inns[string.ascii_letters[i - 1]])
304             # If we've hit the end of the line, go ahead and close
    up the circle.
305             if i == num_verts - 1:
306                 inn.setRightInner(inns[string.ascii_letters[0]])
307 
308         # then make the outer
309         out = Outer(str(i + 1))
310         # Go ahead and connect the inner we just made with this
    outer node
311         out.setAdjInner(inn)
312         # If we aren't on the first one, go ahead and connect it to
    the previous segment
313         if i != 0:
314             out.setLeftSegment(segs[str(-i)])
315 
316         # Now time to make the segment
317         seg = Segment(str(-i - 1))
318         # Go ahead and connect the outer node we just made with this
     segment
319         seg.setLeftOuter(out)
```

```
320         # If we're at the end of the circle, then we close it up.
    Otherwise, move on
321         if i == num_verts - 1:
322             seg.setRightOuter(outs[str(1)])
323 
324         # add them to our dictionaries
325         segs[seg.getName()] = seg
326         outs[out.getName()] = out
327         inns[inn.getName()] = inn
328 
329     # If we've made it here, then we've made the full circle and are
     ready to return it
330     return segs, outs, inns
331 
332 
333 def findTheFace(source_in: Inner) -> list:
334     """
335     This will take an inner node and use the algorithm to walk the
    face that it is on.
336     The order of the face will be i, o, s, o, i repeat
337     :param source_in: Inner node object we are starting from.
338     :return: face: a list representing the face. This list is of
    inner, outer, and segment objects in the
339              order i, o, s, o, i, repeat.
340     """
341     # initialize the list
342     face = list()
343     # starting the face with the source inner node.
344     face.append(source_in)
345     # initialize the ending inner node we will be using for
    comparison
346     end_in = None
```

```
347     # As long as we haven't looped back around, go through the
    following process.
348     while source_in != end_in:
349         # inner: find adjacent outer
350         face.append(face[-1].getAdjOuter())
351         # outer: go to right seg
352         face.append(face[-1].getRightSegment())
353         # segment: go to right outer
354         face.append(face[-1].getRightOuter())
355         # outer: then adj inner
356         face.append(face[-1].getAdjInner())
357         # then left inner and repeat.
358         # set this inner node as our node to compare to our starting
     node.
359         end_in = face[-1].getLeftInner()
360         face.append(end_in)
361     return face
362
363
364 def faceCannonOrder(face: list) -> list:
365     """
366     Just list the face with the face elements in order.
367     We will do it with the first numerical face, and then go right
    before it for an order that will be consistent.
368     :param face: a list representing the face. This list is of inner
    , outer, and segment objects in the
369             order i, o, s, o, i, repeat.
370     :return: ordered face in canonical order
371     """
372     # find the first numerical face then go right before it
373     # initialize face num as a relatively high number we won't
    encounter
```

```
374     facenum = 333
375     # initialize the int for where we will split the list
376     start_ind = 0
377     # loop through and find the face we want to find
378     for i in range(len(face)):
379         try:
380             if int(face[i].getName()) < facenum:
381                 # To get here, we must have found a lower face
382                 # keep track of where this is located in the list
383                 start_ind = i - 1
384                 # make our current lowest face the new lowest face
    to keep comparing to.
385                 facenum = int(face[i].getName())
386         # if we try casting a letter to a number, python will get
    upset, but that also means we're looking at
387         # an inner node, which we don't want for this anyways.
388         except ValueError:
389             continue
390
391     # make our ordered face getting from the starting index to the
    end, then wrapping around and getting the rest of
392     # the face
393     ord_face = face[start_ind:] + face[:start_ind]
394     # go through and make sure we don't have any duplicate elements
    right by each other. If we do, then drop them.
395     for i in range(len(ord_face) - 1):
396         if ord_face[i].toString() == ord_face[i + 1].toString():
397             ord_face.pop(i)
398             break
399
400     # return the ordered face
401     return ord_face
```

```
402
403
404 def grabAllTheFaces(inns: Dict[str, Inner]) -> list:
405     """
406     Function to get the list of unique faces for our circle.
407     :param inns: dictionary of Inner objects. We will loop through
    these to get the faces
408     :return: faces: List of distinct faces in canonical order.
409     """
410     # initialize the list of faces
411     faces = list()
412     # a set of all the elements we have covered by the faces. Will
    use this for a completeness check
413     covered = set()
414     # run through every inner node we've been given
415     for inn in inns:
416         # Generate the face that inner node lies on
417         face = findTheFace(inns[inn])
418         # put the face we've gotten in canonical order
419         face = faceCannonOrder(face)
420         # Check if we've already captured it.
421         if face not in faces:
422             # If not, then add it to our list of faces
423             faces.append(face)
424             # Go ahead and add the elements in this face to our
    covered set
425             covered.update(face)
426
427     # check we've gotten all the elements
428     if len(covered) == (3 * len(inns)):
429         print('We got em!!!')
430
```

```
431     # Now return a list of all the faces we have.
432     return faces
433
434
435 def printCircleStatus(segs: Dict[str, Segment], outs: Dict[str,
    Outer], inns: Dict[str, Inner]):
436     """
437     Helper function that prints the status of the circle to the
    console
438     :param segs: dictionary of str: Segment objects in the circle
439     :param outs: dictionary of str: Outer objects in the circle
440     :param inns: dictionary of str: Inner objects in the circle
441     :return: None
442     """
443     # Run through the segments
444     print('\nSegments:')
445     for k in segs:
446         print()
447         print(k)
448         print(segs[k].toString())
449
450     # Run through the Outer nodes
451     print('\nOuters:')
452     for k in outs:
453         print()
454         print(k)
455         print(outs[k].toString())
456
457     # Run through the Inner nodes
458     print('\nInners:')
459     for k in inns:
460         print()
```

```
461         print(k)
462         print(inns[k].toString())
463 
464 
465 if __name__ == '__main__':
466     # This is where you change the variables.
467     # must be a positive integer > 2
468     verts = 12
469     # Must be a string with spaces between each element. If you want
        to denote multiple cycles, you must add a |
470     switch_txt = '2 3 4 5 | 12 7'
471 
472     # we're going to make a list of all the switches and all the
        cycles
473     switches = list()
474     # first, we get the cycles, split by '|'
475     cycles = switch_txt.split('|')
476     for c in cycles:
477         # We're going to split the switch into a list split by the
        whitespace
478         s = c.strip().split()
479         # Then we're going to append the switches in the cycle to
        the new list
480         switches.append(s)
481 
482     # Go ahead and make the standard circle given the number of
        vertices we want to use.
483     segments, outers, inners = standardCircle(verts)
484 
485     # Go through and grab the faces for our standard circle
486     facs = grabAllTheFaces(inners)
487     print('\nPrinting the faces')
```

```
488     for f in facs:
489         print()
490         for p in f:
491             sys.stdout.write(p.getName() + ' ')
492 
493     # Go through and do the switches for each cycle
494     for switch in switches:
495         for num in range(len(switch)):
496             # store the current part of the switch we're working on
497             cs = switch[num]
498             # store the next part of the switch we're working on,
    looping to the beginning if we're at the end
499             ns = switch[(num + 1) % len(switch)]
500             # Do the actual switch
501             # Getting the new inner and outer validly switched up
502             inners[string.ascii_letters[int(cs) - 1]].setAdjOuter(
    outers[ns])
503             outers[ns].setAdjInner(inners[string.ascii_letters[int(
    cs) - 1]])
504 
505     # print how the final rotation sits
506     printCircleStatus(segments, outers, inners)
507 
508     # Go through and generate and print the new faces
509     new_facs = grabAllTheFaces(inners)
510     print('\nPrinting the new faces')
511     for f in new_facs:
512         print()
513         for p in f:
514             sys.stdout.write(p.getName() + ' ')
```

Bibliography

[1] Ergun Akleman, Jianer Chen, and Jonathan L. Gross. "Extended graph rotation systems as a model for cyclic weaving on orientable surfaces." In: *Discrete Applied Mathematics* 193 (2015), pp. 61–79. ISSN: 0166-218X. URL: https://ezproxy.mtsu.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselp&AN=S0166218X1500195X&site=eds-live&scope=site.

[2] Peter Eades et al. "A linear time algorithm for testing maximal 1-planarity of graphs with a rotation system." In: *Theoretical Computer Science* 513 (2013), pp. 65–76. ISSN: 0304-3975. URL: https://ezproxy.mtsu.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselp&AN=S0304397513007214&site=eds-live&scope=site.

[3] M.N. Ellingham, Chris Stephens, and Xiaoya Zha. "The nonorientable genus of complete tripartite graphs". In: *Journal of Combinatorial Theory, Series B* 96.4 (2006), pp. 529–559. ISSN: 0095-8956. DOI: https://doi.org/10.1016/j.jctb.2005.10.004. URL: https://www.sciencedirect.com/science/article/pii/S009589560500153X.

[4] M. N. Ellingham and Xiaoya Zha. *Orientable embeddings and orientable cycle double covers of projective-planar graphs*. 2009. arXiv: 0911.2713 [math.CO].

[5] Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces.* Johns Hopkins series in the mathematical sciences. Johns Hopkins University Press, 2001, pp. I–XI, 1–291. ISBN: 978-0-8018-6689-0.

[6] Neil Robertson, Xiaoya Zha, and Yue Zhao. "On the flexibility of toroidal embeddings". In: *Journal of Combinatorial Theory, Series B* 98.1 (2008), pp. 43–61. ISSN: 0095-8956. DOI: https://doi.org/10.1016/j.jctb.2007.03.006. URL: https://www.sciencedirect.com/science/article/pii/S0095895607000536.

[7] John Maharry et al. "Flexibility of projective-planar embeddings." In: *Journal of Combinatorial Theory, Series B* 122 (2017), p. 241. ISSN: 0095-8956. URL: https://ezproxy.mtsu.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsgao&AN=edsgcl.472801904&site=eds-live&scope=site.