Delaunay triangulations and combinatorial Ricci flows

Alex Henniges
Thomas Williams
Mitch Wilson

University of Arizona Undergraduate Research Program
Supervisor: Dr. David Glickenstein

August 14, 2008

Table of Contents

1   Introduction

The purpose of this project is to explore and extend upon topics currently under research in Differential Geometry. Concepts will be addressed under a programmable, analytic structure to represent and evaluate Delaunay triangulations and combinatorial Ricci flows. Our goal is to solidify a data structure, as well as confirm known facts and conjecture new theories in this fairly unexplored area of mathematics. Our program is written in C++.

We begin in §2 with an introduction to triangulations and an explanation of how our program is structured to meet its function. We will then present our research on particular types of triangulations, known as Delaunay triangulations, in §3. §4 will follow with an explanation of combinatorial Ricci flow along with results from our experiments. In §5 we continue the exploration of combinatorial Ricci flow under different geometries. Lastly, in §6, we will provide areas of future research towards relating and expanding the two subjects.

2   Triangulations

Suppose you are asked to construct the surface of a sphere with as few pieces as possible. You could make a number of possible shapes, such as a cube or a soccer ball. While both of these shapes are discrete in nature, they can be used to approximate a round, continuous sphere. The most basic Euclidean approximation of the surface of a sphere is the boundary of a tetrahedron. Using only four vertices, six edges, and four faces, the tetrahedron is able to give us a surface that represents the surface of a sphere. Naturally, if we add more vertices, we are able to better illustrate our shapes through greater refinement. Similarly, in modern video games and Hollywood movies, various shapes are generated using polygons of many different sizes that mold to form a graphically rendered object. For these and all shapes, we will focus on representing them solely out of triangles. Since any regular polygon can be broken up into triangles, we can essentially represent any surface using this method.
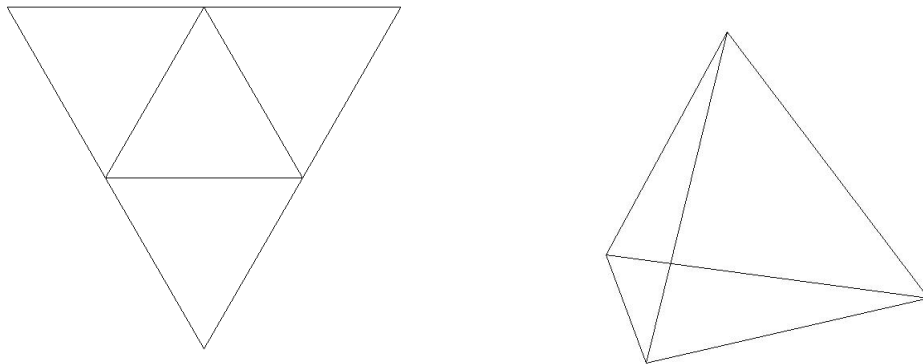
Figure 1: An example of a triangulation. In this case, the triangle, a 2-simplex, can be folded up into the boundary of a tetrahedron, a 3-simplex.

2.1 Basics and Definitions

Definition 2.1. *A triangulation in n dimensions, written $\tau = \{\tau_0, \tau_1, ..., \tau_n\}$, consists of lists of simplices σ^k, where the super-script denotes the dimension of the simplex and τ_k is the list of all k-dimensional simplices $\sigma^k = \{i_0, ..., i_k\}$. We shall refer to 0-dimensional simplices as vertices, 1-dimensional simplices as edges, and 2-dimensional simplices as triangles or faces [4].*

In two dimensions, we think of a triangulation as a surface that is made up of triangles joined along edges. Typically, triangulations are also defined to have no two triangles sharing the same three vertices. We use a more generalized definition that applies to open as well as closed surfaces and allows for shared simplices. In this paper, it will not be necessary to work outside of two dimensions. For that reason, we will refer to the spaces being triangulated simply as surfaces.

In order to define our surfaces we must provide a description for each individual simplex. For our data structure, we decided on creating a list of references to give each simplex within the triangulation. These lists of references for each simplex are refences to other simplices in the triangulation with the property of being local. We define the term "local" differently for each simplex.

- Locality is commutative. That is, if one simplex is local to another simplex, then that simplex is also local to the first simplex.

- We write a vertex as its index i, an edge as a pair of vertices $\{i, j\}$, and a face as a triple of vertices $\{i, j, k\}$. Edges do not have a direction, therefore $\{i, j\}$ is the same edge as $\{j, i\}$.

- An edge $\{i, j\}$ is local to vertices i and j. A face $\{i, j, k\}$ is local to vertices i, j, and k, and edges $\{i, j\}$, $\{i, k\}$, and $\{j, k\}$.

- Similarly, a vertex i is local to all edges that meet at i and all faces that meet at i. An edge $\{i, j\}$ is local to all faces that meet at $\{i, j\}$.

- The number of edges local to a given vertex is also known as the *degree* of that vertex.

- A vertex i is local to all vertices that share an edge with i.

- An edge $\{i, j\}$ is local to all edges local to vertices i and j.

- A face $\{i, j, k\}$ is local to all faces that share edges $\{i, j\}$, $\{i, k\}$, and $\{j, k\}$.

As well as lists of references to other simplices, each simplex also has other information needed to form a triangulation. While we say that these lists of references define the topology of a given triangulation, we must provide dimensions in order to define its geometry. Namely, each edge $\{i, j\}$ is given a length l_{ij} and each vertex i is given what we call a *weight*, denoted by r_i. We think of the weight of a given vertex to be the radius of a circle centered at that vertex. For any n-dimensional simplex, there exists an $(n-1)$-dimensional sphere that is orthogonal to each of the spheres centered at the vertices which define that particular simplex [4]. We use this sphere, and its center, to define the center of the given simplex. For our project, this means that a triangle $\{i, j, k\}$ has a center defined by the center of the circle which lies orthogonally to all of the circles at each vertex. We call this circle the *orthocircle*.

For an edge $\{i, j\}$, it is true that

$$l_{ij} = d_{ij} + d_{ji},$$

where d_{ij} is the *local length* of edge $\{i, j\}$ from vertex i, or the length from vertex i to C($\{i, j\}$), the center of $\{i, j\}$. In this way, we have defined the centers for edges. If we examine the edge lengths of a given triangle, it can be concluded that

$$d_{ij} = \frac{l_{ij}^2 + r_i^2 - r_j^2}{2l_{ij}} \tag{1}$$

It also holds that

$$d_{ij}^2 + d_{jk}^2 + d_{ki}^2 = d_{ji}^2 + d_{kj}^2 + d_{ik}^2 \tag{2}$$

for any face $\{i, j, k\}$. Based on this condition, we are guaranteed that for every face $\{i, j, k\}$, the perpendiculars from each edge center meet at a single point, which is the center of the face $C(\{i, j, k\})$ as defined above [4].

There are two notable examples of choices of weights and their corresponding face centers. The first is when all of the weights are zero. In this case, the triangle center $C(\{i, j, k\})$ is the center of the circumcircle of $\{i, j, k\}$. The second is when the weights are equal to the local lengths for i, or for all vertices j and k that are local to i, we have $d_{ij} = d_{ik}$. This case is known as a *circle packing* and will be used in §4. In this case, the triangle center $C(\{i, j, k\})$ is the center of the incircle of $\{i, j, k\}$.

We will now make definitions for duals of our simplices. First, the dual of a face is simply its center $C(\{i, j, k\})$ as defined above. Now we will define a dual for a given edge. Let us refer to any edge, along with its local faces, as a *hinge*. We are able to embed any hinge in the plane. Because of (2), we know that on any hinge in the plane, the center of an edge along with the centers of its two local faces are colinear. The line connecting these three points is the dual for the given edge. We can determine the length of this dual if we can determine the length of the line segment connecting the center of an edge to the center of one of its local faces. On triangle $\{i, j, k\}$, we write the distance between $C(\{i, j\})$ and $C(\{i, j, k\})$ as

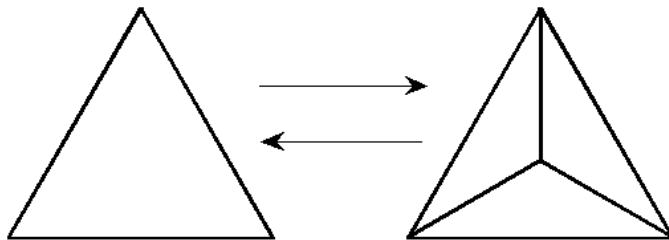$$h_{ij,k} = \frac{d_{ik} - d_{ij}\cos(\theta_i)}{\sin(\theta_i)} \tag{3}$$

7

Figure 2: The 1-3 and 3-1 moves.
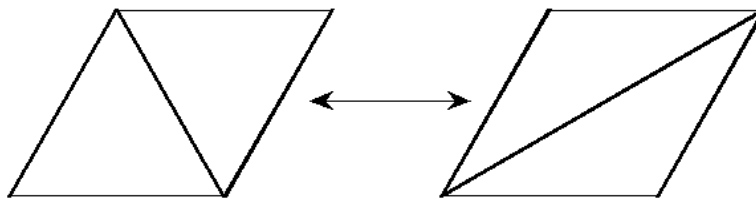
Figure 3: The 2-2 move.

where θ_i is the angle at i on face $\{i, j, k\}$. We refer to this distance as the height of $C(\{i, j, k\})$ with respect to $\{i, j\}$. In the case that $C(\{i, j, k\})$ lies outside of $\{i, j, k\}$ on the side of $\{i, j\}$ where the opposite faces lies, then its height has a negative value. Otherwise, the height is positive [4]. Naturally, the dual of a given edge has a distance that is obained by adding the heights of the centers of the faces local to the edge with respect to that edge. The dual of a given vertex i is the area enclosed by the duals of every edge local to i. The duals of edges will be talked about again in §3.

2.2   Bistellar moves

On a given triangulation, we are able to perform different modifications, which we will refer to as *bistellar moves* or *flips*. There are three possible bistellar moves that can be performed on a 2-dimensional triangulation: a 1-3 move, a 2-2 move, and a 3-1 move. The 1-3 move and the 3-1 move are inverses of one another while the 2-2 move is its own inverse.

For the 1-3 move, we consider a face $\{i, j, k\}$. We create a new vertex l and place it inside of $\{i, j, k\}$. From this, we create edges $\{i, l\}$, $\{j, l\}$, and $\{k, l\}$. By doing this, we have turned $\{i, j, k\}$ into three new faces, $\{i, j, l\}$, $\{i, k, l\}$, and $\{j, k, l\}$ (see fig. 2). The 3-1 move is simply the inverse of this move. We start by taking a vertex l of degree 3 and removing it from the triangulation, essentially removing three faces and three edges with it and leaving behind a single face $\{i, j, k\}$. A 2-2 move is one that requires no additions or removals of any simplices. Consider a hinge with edge $\{i, j\}$ and faces $\{i, j, k\}$ and $\{i, j, l\}$. In a 2-2 flip, we simply take edge $\{i, j\}$ and replace it with a new edge $\{k, l\}$ (see fig. 3). This move will be used extensively in §3.

2.3   Programming Structure

When creating the program, the data structure design was critical. The design not only helps dictate the direction of the project over the course of its lifespan, but the decisions affect the speed and efficiency of all added functionality. Some of the highest priorities of our program are:

- Fast calculations.

- Easy access to data.

- Adaptable to a number of situations.

As seen in Figure 4, all simplices are assumed to have lists of references to their local simplices broken down by dimension. Each simplex has lists of local vertices, local edges, and local faces. The lists are vectors of integers. The vector, provided in the C++ library, was chosen so that the list can dynamically change in size. The integers are a decision based on both speed and size. Instead of, for example, a vertex having a list of actual edges $(\overline{AB}, \overline{CF},$ etc.) or pointers to edges, the vertex has a list of integers representing the edges. The actual edges are then obtained through the $Triangulation$ class, which holds maps from integers to simplices. The $Triangulation$ class consists of static functions and maps and is designed so only one triangulation exists at any time. Because the maps are static, they can be accessed at any time from anywhere in the code without the need to pass pointers through function calls. Maps also have quick access time.

Simplex

-localVertices:vector<int>
-localEdges:vector<int>
-localFaces:vector<int>
-index:int

+getLocalVertices():vector<int>*
+getLocalEdges():vector<int>*
+getLocalFaces():vector<int>*
+getIndex():int
+setIndex(int)
+addVertex(int)
+addEdge(int)
+addFace(int)
+removeVertex(int)
+removeEdge(int)
+removeFace(int)

Line

-x1, y1, x2, y2 : double
-slope : double
-intercept : double
-vertical : bool

+getInitial( ) : Point
+getEnding( ) : Point
+getLength( ) : double
+isAbove(Point) : bool
+hasPoint(Point) : bool

Point

+x : double
+y : double

MiscMath

quadratic(double, double, double):vector<double>
+distancePoint(Point, Point):double
+rotateVector(Point, double):Point
+findPoint(Line, double, double):Point

Vertex

-weight: double

+getWeight():double
+setWeight(double)
+setWeightIndependent(double)

Edge

-length: double

+getLength(): double
+setLength()
+setLength(double)

Face

-negative:bool

+isNegative( ):bool
+switchSide( )

TriangulationCoordinateSystem

-pointTable::map<int, Point>
-lineTable::map<int, Line>
-dualList::vector<Line>

+containsLine(Point)
+containsLine(int)
+generatePlane( )
+update( )
+getPoint(int): Point
+getLine(int): Line
+addHalfDual(Edge, Face)
+addDual(Edge)
+addDuals(Vertex)
+printToFile(char*)
+printDuals(char*)

Triangulation

+vertexTable:map<int, Vertex>
+edgeTable:map<int, Edge>
+faceTable:map<int, Face>

+putVertex(int, Vertex)
+putEdge(int, Edge)
+putFace(int, Face)
+eraseVertex(int)
+eraseEdge(int)
+eraseFace(int)
+netCurvature():Double

TriangulationPlane

+firstTriangle(double, double, double)
+addTriangle(Edge, double, double)
+addTriangle(Edge, Edge)
+getAngleSum(Vertex):double
+generateTriangulation(int)
+generateWeights( )
+flipAlgorithm( )
+weightedFlipAlgorithm( )
+checkTriangle(Edge, double, double)
+makeSpecialCase( )

TriangulationMath

+angle(Vertex, Face): double
+curvature(Vertex): double
+calcFlow(vector<double>*, vector<double>*, double, double*, int, bool)

TriangulationMorph

+addNewVertex(Face, double)
+flip(Edge)
+removeVertex(Vertex)
+addLeaf(Edge, double)

Delaunay

+isDelaunay(Edge):bool
+isWeightedDelaunay(Edge):bool
+getHeight(Face, Edge):double
+getDual(Edge):double
+getPartialEdge(Edge, Vertex):double

SphericalMath

+sphericalAngle(Vertex, Face): double
+sphericalCurvature(Vertex): double
+sphericalCalcFlow(vector<double>*, vector<double>*, double, double*, int, bool)

HyperbolicMath

+hyperbolicAngle(Vertex, Face): double
+hyperbolicCurvature(Vertex): double
+hyperbolicCalcFlow(vector<double>*, vector<double>*, double, double*, int, bool)

TriangulationInputOutput

+readTriangulationFile(char*): bool
+writeTriangulationFile(char*)
+makeTriangulationFile(char*, char*)
+printResultsStep(char*, vector<double>*, vector<double>*)
+printResultsVertex(char*, vector<double>*, vector<double>*)
+printResultsNum(char*, vector<double>*, vector<double>*)
+printDegrees(char*)
+printEdgeLengths(char*)
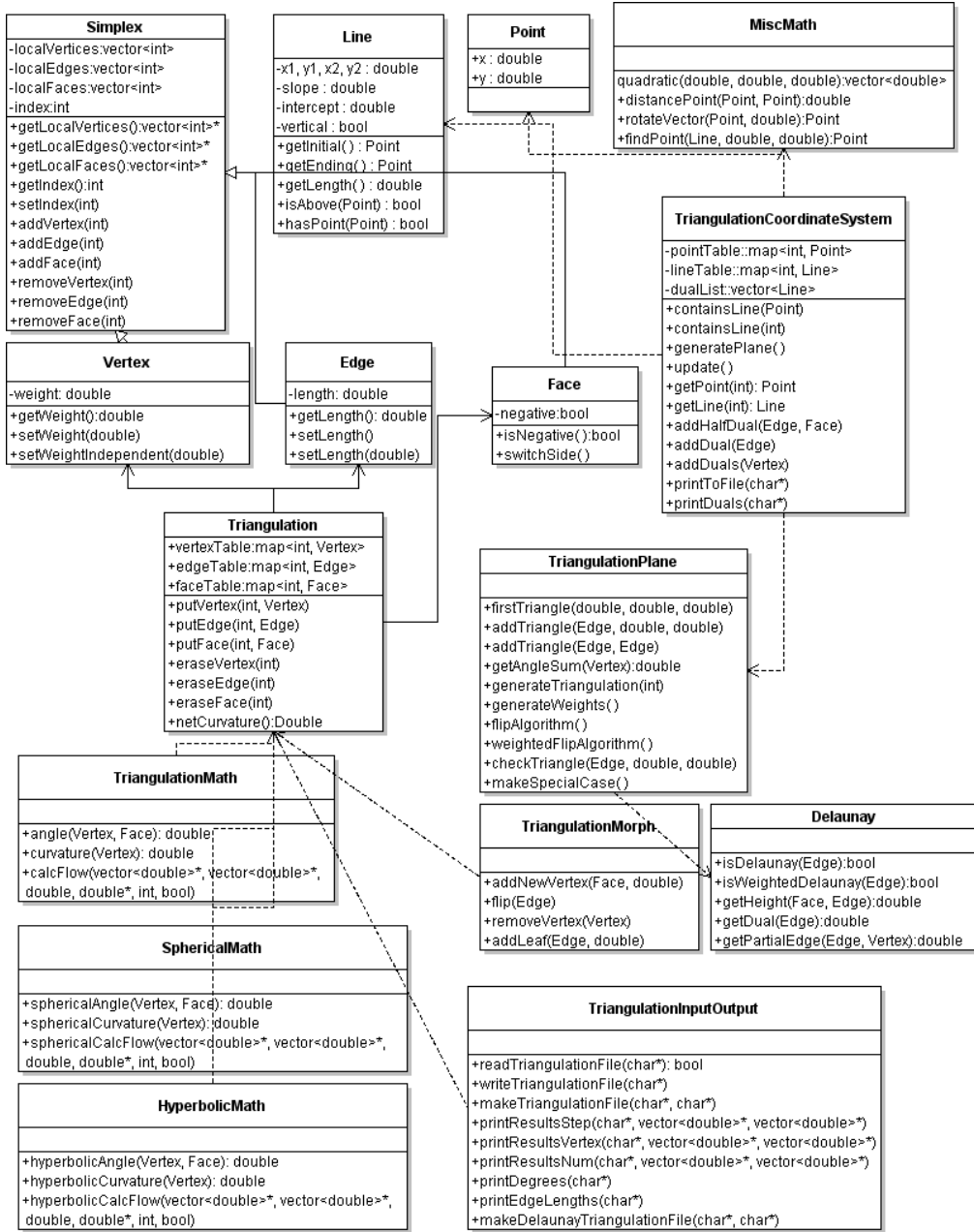+makeDelaunayTriangulationFile(char*, char*)

Figure 4: A UML diagram of the program. The UML shows how the various files interact as well as the functions and variables they hold. We will refer to many of these programs later on.

10

Vertex: 1	Edge: 1	Face: 1
2 3 4	1 2	1 2 3
1 2 4	2 3 4 5 7	1 2 3
1 2 3	1 2	2 3 4
Vertex: 2	Edge: 2	Face: 2
1 3 4 5	1 3	1 2 4
1 3 5 7	1 3 4 6 8	1 4 5
1 2 4 5	1 3	1 3 5
⋮	⋮	⋮
Vertex: 5	Edge: 9	Face: 6
2 3 4	4 5	3 4 5
7 8 9	4 5 6 7 8	6 8 9
4 5 6	5 6	3 4 5

Table 1: The format used to represent a triangulation in our program.

In order to build the triangulations that we need for our tests, we decided upon a format that could be entered into a file and then read by our program. This format can be seen in Table 1. While this format works fine and we can create some basic triangulations by hand, it is not easy for creating a wide array of triangulations.

Frank Lutz, creator of The Manifold Page [5], has a catalog of almost two million known manifold triangulations of varying sizes. However, the format is different than our setup, so we developed an algorithm to take a given triangulation, saved on its own as a text file, and convert it into the form that we use. We were able to transform this

```
{manifold_lex_d2_n5_o1_g0_#1=[[1,2,3],[1,2,4],[1,3,4],
[2,3,5],[2,4,5],[3,4,5]]},
```

which solely documents the faces, into our format. The result is shown in Table 1.

These functionalities of our code form the cornerstone for the rest of the program and have to be the most adaptable to future needs, both seen and unseen. Information on other important aspects, like calculating angles,
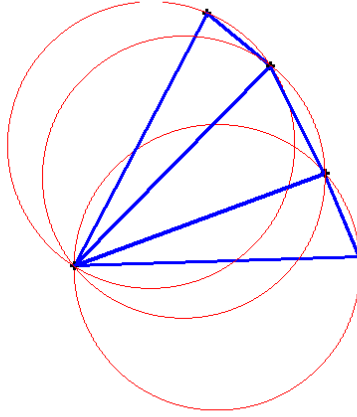
Figure 5: A simple example of a Delaunay triangulation. The circumcircles are drawn along with the edges to show their compatibility. This is a non-weighted case.

displaying triangulations, and useful objects like points and lines can again be found in Figure 4 or in later sections.

3 Delaunay triangulations

3.1 Definitions

One special type of triangulation is known as a *Delaunay triangulation*, of which there are two different types. The first type is the case of non-weighted triangulations, which we simply call "Delaunay", while the second type is reserved for weighted triangulations, which we refer to as "weighted Delaunay". We will refer to the condition of not being Delaunay as non-Delaunay and the condition of not being weighted Delaunay as non-weighted-Delaunay. Before we can define what it means for a triangulation to be Delaunay or weighted Delaunay in a global sense, we must define what it means for a triangulation to have either of these conditions in a local sense, and in order to do that, we must consider a single hinge.

To determine if a hinge is Delaunay, we must consider the orthocircles of both of the triangles in the hinge. In the non-weighted case, this is the circumcircle of the triangle. If the circumcircle of either of these triangles (and as it turns out, necessarily both of them) contains the opposite vertex in the hinge, then we say that the hinge is not Delaunay. However, if both triangles have circumcircles that do not contain the opposite vertex on the hinge, then the hinge is Delaunay. It is worth noting that all non-convex hinges, that is all hinges with one of the angles on the edge having a measure greater than π, are Delaunay.

Proposition 3.1. *The sum of the opposite angles on a non-Delaunay hinge is greater than π.*

Corollary 3.2. *Any non-Delaunay hinge will become Delaunay after a 2-2 flip is performed on it.*

This proof is given in A.2. One thing that is important to note after the discovery of this property is what we refer to as the "boundary case", where the hinge forms a cyclic quadrilateral, i.e. all the vertices lie on the same circle. This occurs when the centers of the triangles, given by the centers of their orthocircles, lie on the same point. In such a case, both the hinge and its flipped image are considered Delaunay.

The condition of a hinge being weighted Delaunay is somewhat different. As stated before, the dual of a hinge is found by connecting the centers of the two orthocircles. Its length can be found by adding up the two heights. Thus, for two faces, $\{i, j, k\}$ and $\{i, j, l\}$, the length of the dual on the hinge is given by

$$\star\{i, j\} = h_{ij,k} + h_{ij,l}$$

Recall that a height is measured as negative if the center of the triangle lies on the opposite side of the hinge. Since one or both heights may be negative, we obtain the following:

Definition 3.3. *In the case of a weighted triangulation, a hinge is weighted Delaunay when the dual has positive length. If the dual has negative length, then it is not weighted Delauany.*
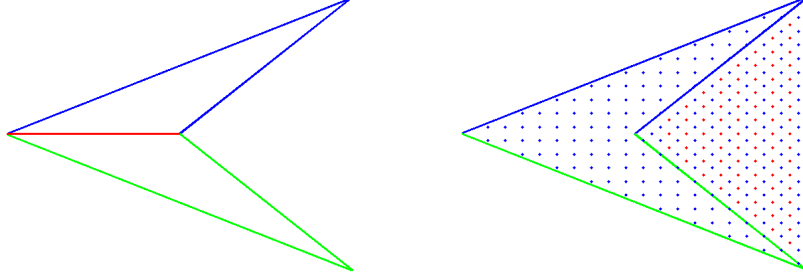
Figure 6: If flips are performed on nonconvex triangles, we create an anti-triangle, shaded here in red, and a larger regular triangle, shaded in blue.

A hinge that is nonconvex can also be not weighted Delaunay. This creates issues when one wishes to perform flips on hinges that are non-weighted-Delaunay. It is not entirely clear how one goes about performing a flip on a hinge that is nonconvex. Figure 6 shows such a hinge before the flip is performed next to the resulting image of a triangle with an "anti-triangle" lying over a portion of it. We consider this triangle to have negative area, its angles to have negative measure, and its heights, when otherwise considered positive, to be of negative length.

There are a number of remarks to be made regarding negative triangles in our triangulations. First of all, we can give all of our triangles a score, based on their positive or negative orientation, of either 1 or -1. Using this, we can create a value for any point covered by the boundary of a given triangulation. Regardless of the number of flips performed, where, or how many negative triangles are created from them, if one were to pick any point within the pre-existing boundary, the value is guaranteed to be one. That is to say, for every negative triangle or set of negative triangles created, there is a positive or set of positive triangles created to completely cover them. Additionally, if there is a nonconvex hinge that lies on the border that is flipped where the newly created edge lies outside of the original boundary, the value for every point in the area between this edge and the original boundary will remain 0. From this, we can conclude that the area of any triangulation is preserved under 2-2 flips.

Proposition 3.4. *Any hinge that is non-weighted-Delaunay will become weighted*

14

Delaunay after a 2-2 flip is performed on it.

A proof for this is given in §*A*.3. From this we have developed a conjecture.

Conjecture 3.5. *Given an arbitrary triangulation in the plane with randomly assigned weights, one can create a weighted Delaunay triangulation through a series of 2-2 flips.*

The goal from this point forward for this part of the project is to write a computer algorithm that effectively asserts this conjecture.

3.2   Programming Structure

We can produce a non-weighted Delaunay triangulation using mathematical software like MATLAB. However, less is known about weighted triangulations in general. For this project, we aim to develop an algorithm that can create a valid triangulation, add weights to vertices, and manipulate the faces to produce a weighted Delaunay triangulation.

An important aspect to properly explore these triangulations is in their generation. We developed a program *generateTriangulation* which can generate a determined number of faces with a random configuration. To truly test our algorithms and to provide convincing results would require an unbiased way of constructing the triangulations. There are several properties to triangulations that we wish to be randomized, and they include the lengths of each edge, the degree of each vertex, and the weight at each vertex. It is not clear to us if there is a way to provide a truly random generation of triangulations or even if such a system would be desirable, but we feel that the algorithm we chose adequately meets our goals.

Our random triangulation generator begins with one triangle with edge lengths between 0.0 and 10.0. This was chosen arbitrarily as all future weights will be based off of these. Triangles are then created from each edge along the border. When the creation of an edge would result in a vertex's angles summing to greater than 2π, we instead close off the vertex by connecting its two outer edges. Once the desired number of faces have been created, the algorithm ends. Figure 7 shows two results of this algorithm.
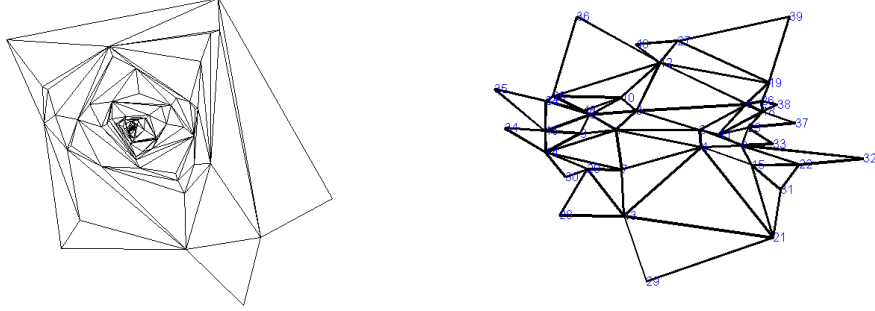
Figure 7: Two examples of triangulations using our *generateTriangulation* program, an early version (left) and a modern version (right). The early version wound up spiraling around itself, increasing edge lengths as it went. Our newer version makes the edge lengths more random, with large and small triangles scattered throughout.

We can generate weights at random by restricting their maximum size with respect to the shortest edge length in the triangulation, so as not to make weights too large. We can also assign weights to individual vertices such that others remain unweighted.

We created a program called *weightedFlipAlgorithm* that scans each edge to see if it is Delaunay, and if not, we perform a flip on the hinge. We repeat the process until (hopefully) each face was Delaunay, and thus the whole triangulation was now Delaunay. Unfortunately, the process did not always end.

To get a better grasp of the sequence of flips, we made a MATLAB program named *delaunayPlot* that draws a triangulation so we could visually ensure that our triangulations are being built correctly, see if and where flips were being performed, and track negative triangles. We use MATLAB to generate pictures relatively easily, including most figures in this paper. We adapted this code to then make *multiDelaunayPlot*, which regraphs the triangulation after a flip is performed. See Figure 8 for a beginning and end triangulation after going through *weightedFlipAlgorithm*.
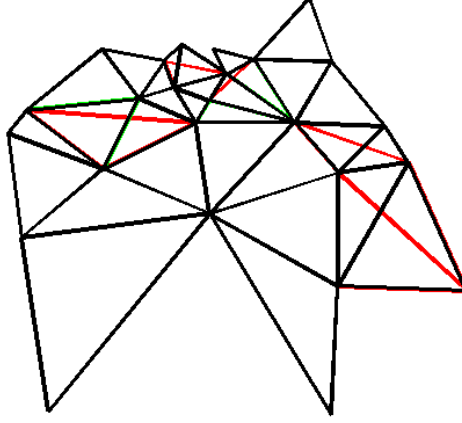
Figure 8: An illustration of our *delaunayPlot* program. The end triangula-
tion is shown in black. Original edges that were later flipped are
shown in red.

3.3 Results

Often, we did find that triangulations became Delaunay after a series of
flips. As we added more faces and increased the weights, we noted that it
was more likely for the algorithm to have issues. See Table 2. In watching
our *multiDelaunayPlot* results, we found that certain edges would loop in
a series of flips such that performing one flip would require an adjacent edge
to need flipping. Occasionally, one edge would get stuck flipping back and
forth between two faces. One thing of note was that final triangulations did
permit the existence of negative triangles. Often they were found on the edge
boundary and of little concern. We were able to check our results by drawing
the dual edges on top of all the faces (Refer to our criteria in §2.1 and §3.1).
If all vertices had zero weight, then the duals are simply the perpendicular
bisectors of each edge. See Figure 9.

We also observed a couple of interesting things from a experimental point
of view. We found that, after flips were performed, the average length of
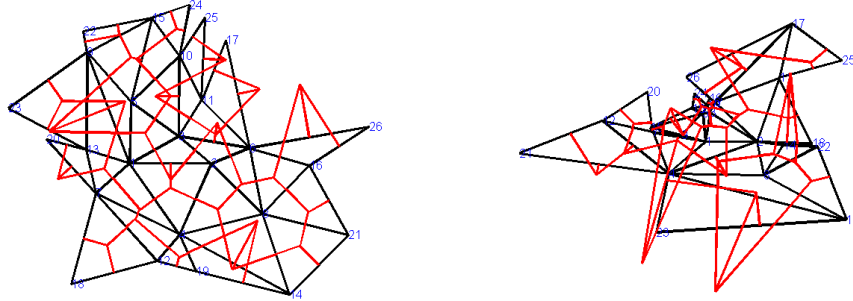all edges tended to decrease by a small amount. See Figure 10 for such

Figure 9: A non-weighted (left) and weighted (right) triangulation with dual lengths added.

Scalar = 1.0			Scalar = 1.5		
Faces	Avg. Flips	% Finish	Faces	Avg. Flips	% Finish
25	4.1	100	25	13.5	83.33
50	11.2	100	50	19.6	71.29
100	26.8	100	100	41.7	40

Table 2: An analysis of number of flips required to obtain Delaunay triangulation for a given number of faces and scaled weights. Each test was performed on *weightedFlipAlgorithm* until each test had ten successful runs. Each trial had unique weights.

an example. There are fewer edges with large weights, and more edges with smaller lengths. This would lead to suggest that vertices try to form triangles with vertices close to itself, resulting in fewer skinny triangles. We also noted that the degrees of vertices sometimes vary more, especially in weighted triangulations; more vertices have a degree of 2 or 3, while others increase in degree. This suggests that the weights of vertices in its triangulation also have a great effect as well as their relative proximities.

In testing our code, we used oneparticular example to examine a potential special case that may have occurred during the running of our algorithm. The special case, as shown in Figure 11, is one where every convex hinge is weighted Delaunay and every nonconvex hinge is not. The results of running our algorithm on this triangulation were of great interest and helped us in
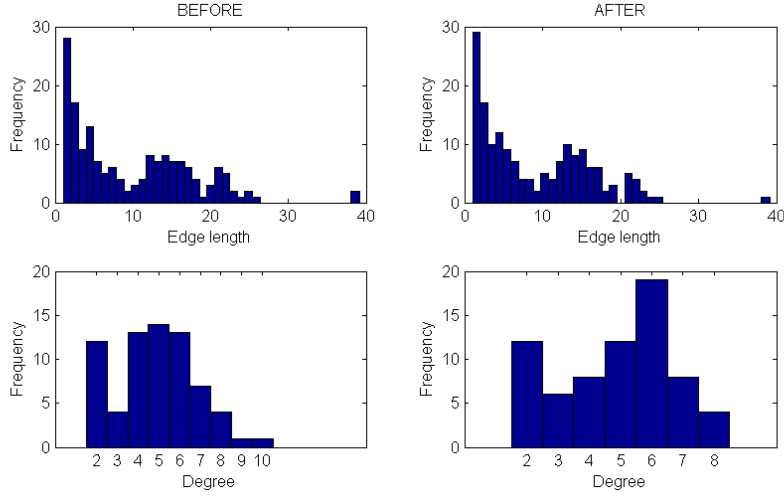
Figure 10: A statistical illustration of edge lengths and vertex degrees before and after flips are performed.

further refining the algorithm. When we started, there were seven faces, all positive. When the flip algorithm concluded, we were left with four positive faces and three negative faces. All of the negative faces were paired with positive faces defined by the same three vertices. The remaining odd positive face was defined by the three original boundary edges. This meant that by "flipping through" all of the nonconvex hinges, we were left with three pairs of reversely oriented faces, what we also referred to as "flaps" or "double triangles".

The hypothesis was that after any triangulation underwent the transformation of the flip algorithm there would remain only positive triangles or negative triangles that sat opposite of positive triangles lying in the exact same area. The algorithm was modified to simply remove these flaps as a last step to the flip algorithm. This process required removing all vertices that lied at the end of a flap, as well as the edges along these flaps. Finally, the two edges defined by the same two vertices from which these flaps were stemming would be joined into one.
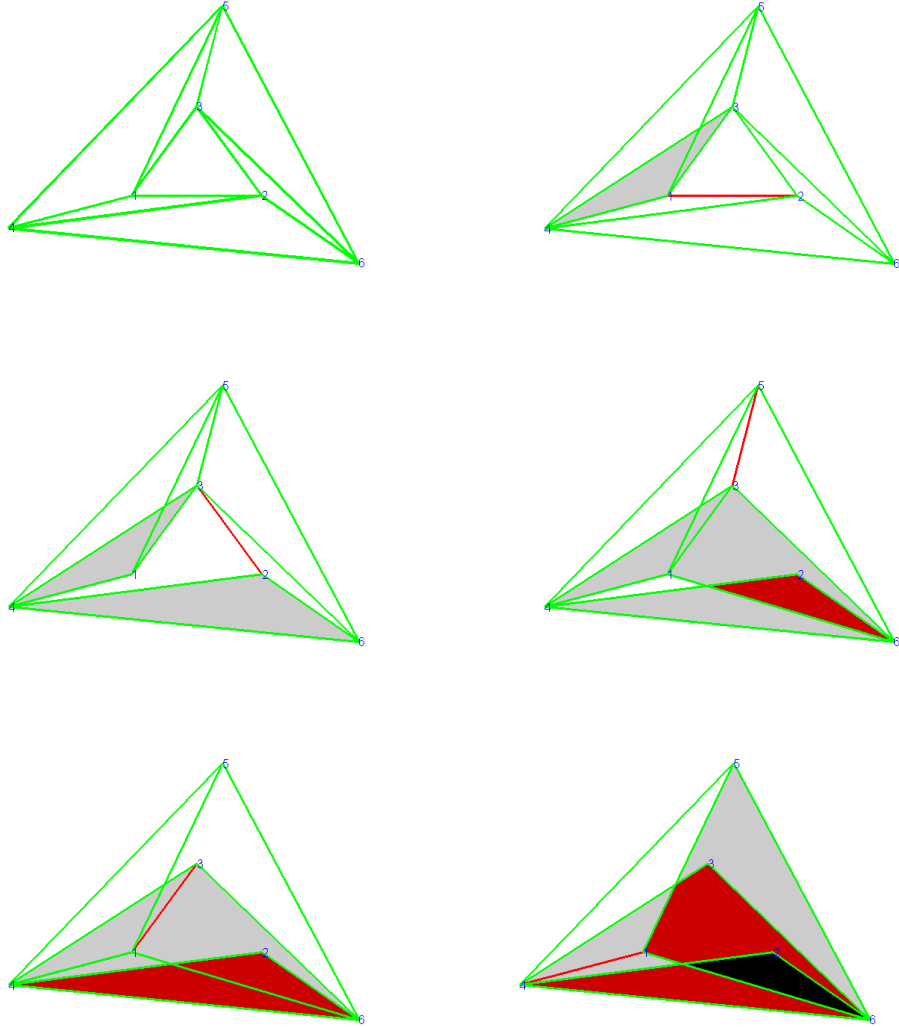
Figure 11: Evolution of our special triangulation. Edges that are flipped are shown in red. Negative triangles start off gray, with intersections a reddish brown. If multiple negative triangles intersect one region, we color it black.

4 Combinatorial Ricci flow

Introduced by Richard Hamilton in 1982, Ricci flow, named in honor of Gregorio Ricci-Curbastro [6], has since had a large influence in the world of geometry and topology. It is often described as a heat equation on geometries. Imagine a room where a fireplace sits in one corner and a window is open on the other side. The heat will diffuse through the room until the temperature is the same everywhere. With Ricci flow, the same occurs with the curvature. Under Ricci flow, a geometric object that is distorted and uneven will morph and change as necessary so that all curvatures are the same. The biggest consequence of Ricci flow came when Grigori Perelman proved the Poincaré Conjecture in 2002. The Poincaré Conjecture, proposed in 1904, was particularly difficult to prove, and was one of seven millennium puzzles. Ricci flow turned out to be the cornerstone for the proof. In addition, its relation to the heat equation may open new doors for work in many fields such as fluid dynamics and general relativity [6]. In 2002, Chow and Luo introduced the concept of combinatorial Ricci flow. They showed that this new concept, performed on a triangulation of a manifold, had many of the properties of Hamilton's Ricci flow. The fact that the subject is still very new makes this research project exciting.

4.1 Circle-packing

Looking back at our weighted Delaunay triangulations, we had the condition that the partial edge lengths were related by Eq. (2). As mentioned in §2.1, we will consider circle-packing. Visually, circle-packing entails placing circles with their centers on a vertex so that neighboring circles are tangent to each other. That is, they intersect at only one point, as shown in Figure 12. Thus, we have the restrictions that $d_{ij} = d_{ik} = r_i, d_{ji} = d_{jk} = r_j$, and $d_{ki} = d_{kj} = r_k$, ensuring that (2) holds.

Definition 4.1. *A Circle Packing Metric is a distance metric that is an assignment of lengths l_{ij} to all edges $\{i, j\}$ in the triangulation and an assignment of weights r_i (the radius of a circle) for each vertex i such that $r_i + r_j = l_{ij}$.*

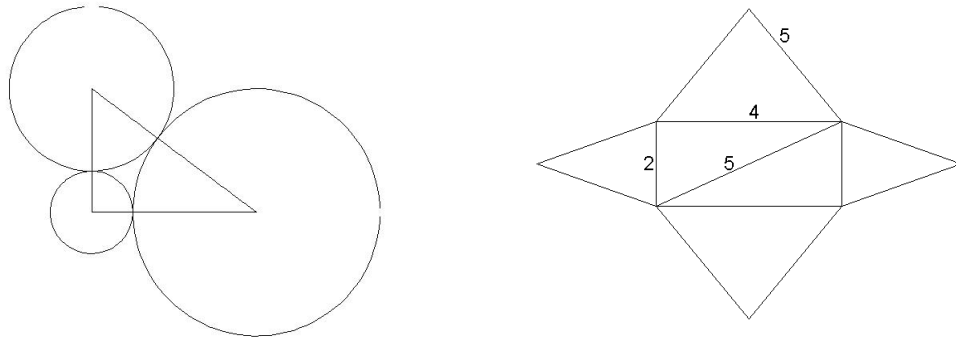Choosing to define the geometry in this way, we guarantee the triangle in-

Figure 12: An example of circle packing (left) and a triangulation that cannot be circle-packed.

equality and ensure that we are creating triangles. Note that not all combinations of edge lengths are possible under this system. Some triangulations can simply not be circle-packed. An example is shown in Figure 12, where there is no combination of weights at the vertices that can make a proper circle-packing. By allowing generalizations of circle-packings, as discussed in §6.2, we can broaden our range of possible length assignments.

4.2 Definitions and Equations

Definition 4.2. *A manifold is a collection of points* $\mathbb{P}$ *such that the neighborhood at every point* $p \in \mathbb{P}$ *is homeomorphic to the open disk.*

What this definition means is that around any point of the manifold it appears similar to the sphere. In particular, there are no borders. We refer to this as a closed surface.

When referring to closed, triangulated surfaces, one important property related is the Euler characteristic χ, defined by $\chi = V - E + F$, where V is the number of vertices in the triangulation, E is the number of edges, and F is the number of faces. This characteristic is directly connected to another important property, the *genus* of a surface. The genus of a surface is the property that is more loosely known as the number of "holes" in the surface.

Name	Vertices, V	Edges, E	Faces, F	Genus, g	χ
Tetrahedron	4	6	4	0	2
Octahedron	6	12	8	0	2
Icosahedron	12	30	20	0	2
Torus	9	27	18	1	0
Two-holed torus	10	36	24	2	-2

Table 3: Listings of vertices, edges, faces, and genus for some common shapes

For instance, a sphere has a genus of 0 while a torus has a genus of 1, a two-holed torus has genus 2, etc. The relationship between the two values is given by $\chi = 2 - 2g$, where g is the genus of the surface.

Once we have a triangulation and it is given a Circle Packing Metric, our manifold may not be in a maximally symmetric geometry. We would like to make the geometry uniform. We introduce combinatorial Ricci flow. This equation allows the weights to change over time [2]. The combinatorial Ricci flow is

$$\frac{dr_i}{dt} = -K_i r_i \tag{4}$$

where K_i is the *curvature* of a vertex, and r_i is the *weight* of the vertex i. The value of K_i changes with time. We take the sum of all angles associated with a vertex i and define the curvature K_i as

$$K_i = 2\pi - \sum \angle i. \tag{5}$$

Using side lengths we can determine the angle using the law of cosines. For a triangle with lengths $a, b,$ and c, the angle opposite side c is

$$\angle C = \arccos\left(\frac{a^2 + b^2 - c^2}{2ab}\right) \tag{6}$$

with similar formulas for the other angles. We see that the system given by 4 is highly nonlinear. This equation often results in weights that decrease to zero. Take, for example, a simple tetrahedron with all sides of equal length. We find that the curvature of each vertex always equals π. Thus in solving the

differential equation computationally we would decrease each vertex weight by the same amount, but the curvature of each vertex would still remain π. This is because, in the Euclidean case, the curvatures are not affected by uniform scaling. Under (4), the radii would continue to decrease until they approach zero length. We have to address this issue since computers don't like working with numbers near zero, as in the denominator of 6. Numerical instability may occur. To avoid this issue, we resize the length of each radius by a scalar, α. We denote each scaled length by $\tilde{r}_i$, i.e., as

$$\tilde{r}_i = \alpha r_i.$$

Each $\tilde{r}_i$ would have its own $\tilde{K}_i$, but since we are scaling all sides by the same factor, this does not effect the curvature of the surface, so $\tilde{K}_i = K_i$. Thus in plugging $\tilde{r}_i$ in to the differential equation we get

$$
\begin{aligned}
\frac{d\tilde{r}_i}{dt} &= \frac{d(\alpha r_i)}{dt} = \alpha\frac{dr_i}{dt} + r_i\frac{d\alpha}{dt} \\
&= -\alpha K_i r_i + \frac{\tilde{r}_i}{\alpha}\frac{d\alpha}{dt} \\
&= -\tilde{K}_i\tilde{r}_i + \frac{\tilde{r}_i}{\alpha}\frac{d\alpha}{dt}.
\end{aligned}
\tag{7}
$$

We also note that $\frac{1}{\alpha}\frac{d\alpha}{dt} = \frac{d(\log\alpha)}{dt}$ using a basic chain rule. In order to find an appropriate value for α we decided to use the following criterion:

$$f(\tilde{r}_1, \tilde{r}_2, \ldots, \tilde{r}_n) = \prod \tilde{r}_i = \prod \alpha r_i = C, \text{ a constant.} \tag{8}$$

We will call this value the *product area* of the surface. This area prevents all radii from decreasing to zero at the same time. By taking the derivative of Eq. (8) with respect to time we find that

$$\frac{d(\log\alpha)}{dt} = \frac{\text{sum of all curvatures}}{\text{number of vertices}} = \overline{K}, \text{average curvature.} \tag{9}$$

In this paper, we may refer to the sum of all curvatures as the *total curvature*. We can also show that $\overline{K}$ is a constant and depends on the number of vertices

24

and the Euler characteristic. We know that the sum of angles on each face is π, thus we determine that

$$\sum K_i = 2\pi V - \pi F = 2\pi(V - \frac{F}{2})$$

We can simplify this by noting some basic facts about triangulations of closed surfaces. As every face is made up of three edges, and each edge belongs to two faces, we can see that $3F = 2E$, or $E = \frac{3F}{2}$. Looking back at Table 3 we note that this is true for all polyhedra listed. Now we rewrite the above equation as

$$\sum K_i = 2\pi(V - \frac{F}{2}) = 2\pi(V - \frac{3F}{2} + F) = 2\pi(V - E + F) = 2\pi\chi.$$

Thus we find that $\overline{K}$ is just $\dfrac{\sum K_i}{|V|} = \dfrac{2\pi\chi}{|V|}$ where $|V|$ is the number of vertices. This is also noted in [2]. Plugging this information back into Eq. (7) we determine that

$$\frac{d\tilde{r}_i}{dt} = -\tilde{K}_i\tilde{r}_i + \overline{K}\tilde{r}_i = (\overline{K} - \tilde{K}_i)\tilde{r}_i \tag{10}$$

However, since everything is now a function of $\tilde{r}_i$ and not α, we can just as easily plug r_i back in to the differential equation instead of $\tilde{r}_i$, so we end up with:

$$\frac{dr_i}{dt} = (\overline{K} - K_i)r_i \tag{11}$$

This is known as normalized Ricci flow, as discussed by Chow and Luo [2]. In the case of our basic tetrahedron from earlier, the radii would not change after each iteration as $\overline{K} = K_i = \pi$ and thus $\dfrac{dr_i}{dt} = 0$ for $i = \{1, 2, 3, 4\}$.

Step 1	Weight	Curv	Step 50	Weight	Curv
Vertex 1:	6.000	0.7442	Vertex 1:	4.557	0.008509
Vertex 2:	3.000	-1.122	Vertex 2:	4.530	-0.01185
Vertex 3:	3.000	-1.373	Vertex 3:	4.534	-0.009091
Vertex 4:	8.000	1.813	Vertex 4:	4.563	0.01268
Vertex 5:	6.000	1.227	Vertex 5:	4.550	0.002772
Vertex 6:	2.000	-3.046	Vertex 6:	4.527	-0.01455
Vertex 7:	4.000	-0.3045	Vertex 7:	4.541	-0.003563
Vertex 8:	8.000	1.989	Vertex 8:	4.559	0.01018
Vertex 9:	5.000	0.07239	Vertex 9:	4.553	0.004906

Table 4: Two steps of a Ricci flow

4.3   Programming Ricci flow

The function *calcFlow* runs a combinatorial Ricci flow on a triangulation and records the data in a file. The algorithm for solving the ODE, provided by J-P Moreau, employs a Runge-Kutta method of order 4 [7]. First, the file name for the data is provided. Then, a *dt* is given by the user that represents the time step for the system. The next parameter is a pointer to an array of initial weights to use. This is followed by the number of steps to calculate and record. Lastly, a boolean is provided, where *true* indicates that the normalized differential equation, (11), should be used. Otherwise, the standard equation (4) is employed. Each step, with every vertex's weight and curvature at that step, is printed to the file. An example is shown in Table 4.

After the initial design of *calcFlow*, tests were run to determine its speed. The time it took to run was directly proportional to the number of steps in the flow. However, it was also proportional to more than the square of the number of vertices of the triangulation. As a result, while a four-vertex triangulation can run a 1000 step flow in three seconds, it would take a twelve-vertex triangulation 43 seconds to run the same flow. After inspecting the speed of the non-adjusted flow in comparison, it became clear that the calculation of total curvature, which should remain constant in two-dimensional Euclidean manifold cases, was being calculated far too often. After being adjusted so that it is calculated just once per step, the speed of

the flow is much faster so that a four-vertex system with 1000 steps takes just one second and twelve vertices is much improved with a time of only four seconds. The code for the *calcFlow* function can be found in §*A.6*.

4.4   Initial testing and results

Initial checks for correctness:

1. Boundary of tetrahedron will converge to equal weights.

2. Constant *product area*.

3. Constant total curvature.

We have some expectations for combinatorial Ricci flow on two-dimensional surfaces. Cases like the boundary of the tetrahedron can be calculated by hand so it is useful to test our results against these surfaces. For example, we expect that the tetrahedron under the standard combinatorial Ricci flow (4) will have all weights converge to zero. Whereas under equation (11) the weights are expected to converge to positive constants.

There are other expected behaviors. We expect that the *product area* will be constant at any intermediate step of the flow. Also, it is predicted that the total curvature of a 2-manifold surface should remain a constant determined by its genus.

Our goal is to create a program that can be built upon later to provide further functionality and options without requiring widespread and time consuming changes to the code. While we certainly expect a number of bugs, we plan to develop methods to test and find any errors in our code.

4.4.1   First tests

The first test was the boundary of the tetrahedron. In the standard equation, it is easily shown that all weights approach zero exponentially fast. We then use the normalized Ricci flow for the rest of the testing. In the normalized equation the tetrahedron's weights approach a single positive number, the fourth root of the *product area* of the tetrahedron, so that the area remains

constant. In addition, all the curvatures converged to the same value, in this case π, so that the total curvature is 4π. Results were similar for the other platonic solids.

The next test was the torus with the standard nine-vertex triangulation. Again all the weights approached the same positive number to maintain constant area. As expected, the curvatures all went to zero while the total curvature remained zero throughout. Further simple tests included triangulations of larger genus, and in all cases the total curvature remained at the constant multiple of π that was expected and the *product area* was also constant. In addition, there does not appear to be any further effect from the initial weights other than determining the area of the triangulation. It is not clear from any of the tests we ran that having large disparities betweeen the weights (some small while others very large) causes a different end result.

In each case all vertices converged to the same curvature. This was not always the situation for the weights, as in many triangulations there would be several final weights. It became clear that this would occur when vertices had different degrees. In fact, we guess that there is a formula relating the *product area* of a weighted triangulation and the degree of a vertex to that vertex's final weight. In most of the examples we tried, when two vertices had the same degree, they had the same final weight, but this is not always the case. One such example is adding three vertices to one face of the tetrahedron. As seen in Table 5, vertices 4, 5, and 6 all have degree four. Yet vertex 5 has a greater final weight than the other two. We believe this is because the local vertices of 5 are different in degree from those of 4 and 6. That is, the degrees of the local vertices of 5 are never less than four, while vertices 4 and 6 each have a local vertex with degree 3. See Figure 23 in the Appendix for an illustration of weights over time.

4.4.2   Morphs and specific cases

Another part of the code is a file strictly made up of functions used to manipulate and alter the triangulations. These functions, known within the code as *morphs*, allow us to change the triangulation in different ways, geometric as well as topological, in order to provide us with different kinds of discrete manifolds over which to run our flow.

Vertex: 1	Vertex: 5
2 3 4 5 6 7	1 2 4 6
1 2 3 7 10 13	7 8 9 12
1 2 3 5 7 9	5 6 7 8
Vertex: 2	Vertex: 6
1 3 4 5 6 7	1 2 5 7
1 4 5 8 11 14	10 11 12 15
1 2 4 6 8 10	7 8 9 10
Vertex: 3	Vertex: 7
1 2 4	1 2 6
2 5 6	13 14 15
2 3 4	1 9 10
Vertex: 4	Edge: 1
1 2 3 5	:
3 4 6 9	:
3 4 5 6	

Final weights for a random initial weighted Triangulation
Vertex 1: 15.692
Vertex 2: 15.692
Vertex 3: 6.18421
Vertex 4: 9.6524
Vertex 5: 10.6409
Vertex 6: 9.6524
Vertex 7: 6.18421

Table 5: Adding three vertices to a Tetrahedron

One such example of our morphs is the use of flips, as mentioned in §2.1. We can perform 1-3, 2-2, and 3-1 flips on given triangulations and observe how those alterations affect the end flow. Flips have the special property that they do not change the Euler characteristic of the surface they are performed upon. For example, in performing a 1-3 flip, we have a total of 3 new edges, 2 more faces, and 1 new vertex. The value of χ does not change as $(V+1)-(E+3)+(F+2) = V-E+F = \chi$. This means we could do any number of these flips without changing the topology of the surface.

In addition, there are other "morphs" that can be performed to change the topology of a triangulation in order to give us some more diverse and interesting shapes over which to run these flows and observe their results. There are two such moves that we can perform.

- Adding Handles By adding a handle to a surface, we are in essence adding a hole to it. One simple way to obtain this result is to "patch" the surface of a torus to the surface that we are working with. This is exactly what we have done. The method removes a face from the existing triangulation and replaces it with a 9-vertex triangulation of a

torus. By using the three existing vertices and edges, we are actually adding 6 new vertices, 24 new edges, and 17 new faces. This operation changes the topology of the triangulation that we are working with. It will reduce the Euler characteristic, χ, by 2 and thus change the total curvature of the surface.

- Adding Cross-Caps The purpose of adding a cross-cap to a manifold is to give us a property of non-orientability. A surface of this kind is characterized by the ability to move an object along it in such a way that it reaches its original location but in mirror-image form. In this way, the portion of the surface known as the cross-cap is, itself, topologically equivalent to the mobius band. It is sufficient to add only a single cross-cap to any given surface,. In conjunction with the method of adding handles, this technique allows us to create any topology in two dimensions.

We introduce here the concept of a *double triangle*. A double triangle is made of two triangles with the same three vertices. As part of a triangulation it can be imagined as flattening a party hat and attaching it to an object along the open end. Allowing for double triangles does not fit under our strict definition of triangulation, which does not allow for degree two vertices. However, we wish to observe how they are affected with combinatorial Ricci flow.

We investigate our programs for sanctity by performing morphs on given manifolds and observe any changes in behavior. We include some examples.

Example: Adding a vertex to a one-holed torus. See figure 13.

By inserting a vertex within a triangulation for a torus, we are essentially creating a bump on the torus. We can then observe what happens as we run it through our *calcFlow* program. The new vertex shrinks in size to a weight much smaller than the other vertices, but to a positive constant. To counter this, the other vertices grow slightly to maintain Eq. (8). In all cases (where the vertices began with different weights), the weight that the new vertex converges to is in a proportion to the other weights by a factor of $3 + 2\sqrt{3}$, the exact proportion necessary to maintain equal curvature amongst all the weights. An oddity here is that the three vertices local to this added vertex converged to the same weight as all the vertices not near the flip, despite a difference in degree.
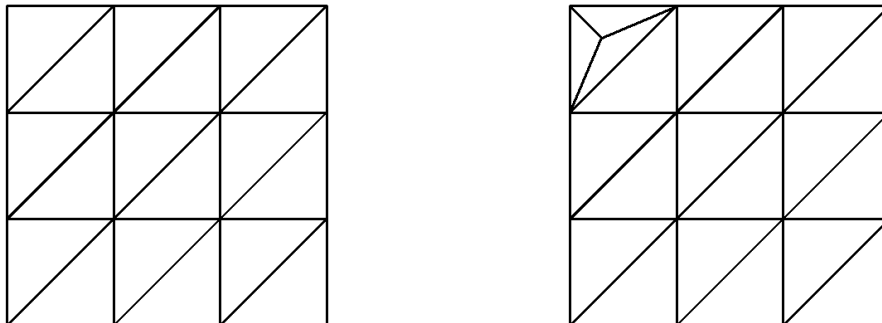
Figure 13: A triangulation of the torus, and the addition of a new vertex. This is a 1-3 flip.

Example: Adding a double triangle to a two-holed torus.

When we added a double triangle to the edge of a two-holed torus, we experienced for the first time what is known as a singularity. At the special vertex that was only of degree two, its weight continued to shrink and never converged. Eventually, enough steps of the Ricci flow were performed that the size of the weight became less than what the computer could differentiate from 0 and the program crashed with a division-by-zero error. Before the crash, the other vertices were increasing without convergence to counteract the decreasing weight. The reason is that all vertices wanted to attain the same curvature, in this case $-\frac{2}{5}\pi$. Yet the new vertex, with only two angles in its sum, can only obtain a curvature of 0 (each angle $\sim \pi$ radians). The result is that the weight shrinks in an attempt to attain angles greater than π, which is simply not possible. What is still not clear is whether or not the weight reaches 0 in finite time, or simply approaches 0. This is difficult to test with a computer only capable of approximating the weight, though we expect that it does so in finite time.

Example: Performing a 2-2 flip on a 12-vertex torus.

Flips can drastically change the behavior of some triangulations. For example, in a 12-vertex torus, performing a flip on a particular edge created a double triangle. Similarly to the previous case, all curvatures are converging to 0, but the vertex with degree 2 simply cannot accomplish this. However,

unlike the previous case, we suspect that the weight does not become 0 in finite time, but instead approaches 0 in infinite time.

Example: Triangulation of genus 4.

While the previous two cases were quite interesting, we were using double triangles and being less restrictive in what were allowable triangulations. But in the same sense that a degree two vertex could have a curvature no less than 0, a degree three vertex is bounded below by $-\pi$. It was found that the curvatures of a triangulation are dependent on the number of vertices and the genus, so a triangulation with a large enough genus and few vertices could have its curvatures converge to less than $-\pi$. The first we found, provided by [5], was an 11 vertex triangulation with genus 4. To create a vertex with degree 3, we chose to add a new vertex to a face with a 1-3 flip. This made all curvatures attempt to converge to $-\pi$. The new vertex reacted as in the previous example, approaching a weight of 0. Whether or not this situation is possible for a vertex of any degree is unknown. To test this one would need a very high genus, but this requires more vertices. Unfortunately, [5] does not provide large enough triangulations to test for vertices with degree greater than three.

Example: Two tetrahedra connected at a vertex. See Figure 14

It turns out $\chi = 7$ Vertices - 12 Edges + 8 Faces = 3, which is not a case we had seen before. Technically, this example is not a manifold, but we felt it was useful in validating our program procedure. Starting with equal weights, we obtained a somewhat unexpected result. The center weight becomes very large, and the others become smaller in comparison. We concluded that since the central vertex has a much higher degree, its weight becomes large, creating elongated tetrahedra on either side, in order to have the same curvature as all other vertices.

4.4.3   Convergence speeds

One experiment we performed was measuring convergence speeds of various triangulations. For each triangulation, five flows were run where the weights were random between one and twenty-five. Random weights, while not a perfect solution, was the most effective option and easiest to implement. As we do not want weights to be a factor in our convergence speed tests,
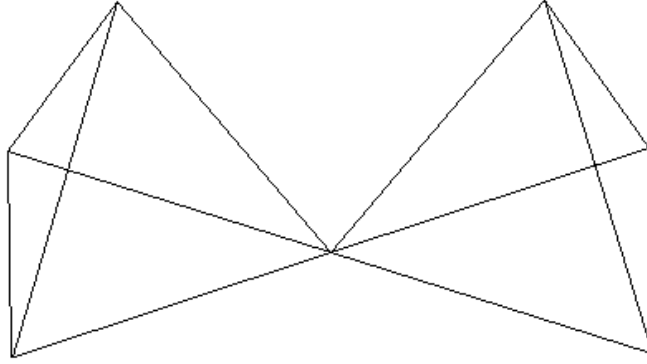
Figure 14: Two tetrahedra joined at a vertex.

it was unclear which set weights could have the same effect for different triangulations. By performing five trials and using random weights, we hope to negate any effect the weights could have on the convergence speed of a triangulation. The *dt* was held constant at 0.03 so that the number of steps needed to run was not large and still provided accurate results. For each run, a step number was assigned for when all weights and curvatures had converged to four digits, (the precision shown in a file of the results).

Beginning with the basic case, the tetrahedron took an average of 156.8 steps to converge to four digits and remained fairly consistent through all five trials. Strangely, the octahedron converged faster on all five flows, averaging 138.4 steps. This was made all the stranger by the fact that the icosahedron averaged 198.8 steps. The torus revealed several things about convergences. First, the standard nine vertex torus averaged only 110 steps, suggesting that a torus converges faster than a sphere. When a vertex was added to the torus, it greatly decreased the convergence speed, to 248.8 steps. Compared to the Tetrahedron with an added vertex, 164.2, this was a very large jump. When another vertex was added to the same face as the first, the convergence speed dropped yet again, to an average of 437.6 steps. Yet when this vertex was added to a face not connected to the first, the convergence speed was

Using dt = 0.03			Random between 1 - 25		
Triangulation	Trial	Steps to converge to four digits	Triangulation	Trial	Steps to converge to four digits
Tetrahedron	1	159	Octahedron	1	127
	2	166		2	153
	3	157		3	134
	4	151		4	136
	5	151		5	142
	Average:	156.8		*Average:*	138.4
Icosahedron	1	217	Torus-9	1	112
	2	179		2	112
	3	209		3	109
	4	172		4	108
	5	217		5	109
	Average:	198.8		*Average:*	110
Tetrahedron with added vertex	1	181	Octahedron with added vertex	1	230
	2	156		2	232
	3	156		3	234
	4	131		4	193
	5	197		5	243
	Average:	164.2		*Average:*	226.4
Torus-9 with added vertex	1	265	Tetrahedron with two added vertices on face 1	1	202
	2	181		2	232
	3	254		3	224
	4	275		4	227
	5	269		5	190
	Average:	248.8		*Average:*	215
Torus-9 with two added vertices on face 1	1	433	Torus-9 with two added vertices on face 1 and 5	1	264
	2	421		2	245
	3	468		3	244
	4	442		4	269
	5	424		5	302
	Average:	437.6		*Average:*	264.8
Torus-9 with three added vertices on face 1 and 5 and 9	1	272	Tetrahedron with two added vertices on face 1 and 2	1	189
	2	270		2	183
	3	297		3	174
	4	291		4	190
	5	231		5	232
	Average:	272.2		*Average:*	193.6
Tetrahedron with flip	1	209	Octahedron with flip	1	200
	2	299		2	210
	3	204		3	225
	4	281		4	182
	5	262		5	237
	Average:	251		*Average:*	210.8
Torus-9 with flip on edge 1	1	107	12-vertex sphere with all different convergences	1	565
	2	131		2	373
	3	134		3	486
	4	116		4	503
	5	138		5	373
	Average:	125.2		*Average:*	460

Figure 15: Summary of convergence data for varying criteria

almost steady at 264.8. And adding a third in the same style caused little increase.

This seems to suggest that convergence speed is dependent on the number of vertices with the same properties. By the properties of the vertex, we mean the number of local vertices, the weight it converges to, etc. When the vertices were added to separate faces, there remained only three unique vertices. Whereas, adding the two vertices to the same face created five unique vertices. This theory is further supported by a twelve vertex sphere designed so that all vertices are unique. The average convergence was 460 steps, more than twice as long as the icosahedron, which also is a twelve vertex sphere.

As a final note, the deviation of the initial weights did not have a clear impact on the convergence speed. At some times, it would appear that initial weights with a higher deviation would converge faster, yet at other times it was lower deviation that seemed to lead to faster convergence.

5   Spherical and hyperbolic Ricci flow

So far we have focused exclusively on triangulations using Euclidean geometry. While this is useful for most cases, there are others where a different background space may be better suited. Two of these in particular are spherical and hyperbolic geometries. We will introduce the basics of each system to readers as they may be unfamiliar with these geometries. We will examine the adaptations of combinatorial Ricci flow for each case, test over various triangulations, and attempt to reach conclusions on our findings.

5.1   Spherical flow

If you were to draw a large triangle on the surface of the earth, you would see that the line segments are no longer linear, but follow along an arc. In dealing with spherical geometry, we learn that many rules of planar geometry, some of them fundamental, do not apply. To begin, the sums of angles in a triangle do not add to 180°. The sum changes depending on the area of the triangle. For computational reasons, we cannot (easily) have edges

of unbounded length. While we can assume our circle packing metric still holds, we do have additional restrictions:

- All triangles must be realizable on a sphere of radius 1, the unit sphere.

- The sum of the weights of any triangle cannot exceed π. As such, no two weights can add up over π. By circle-packing, each edge length must then be less than π, therefore we imply the geodesic between any two points, using the shortest distance on the surface between the two vertices.

We also have different formulas for numerous things in the spherical case. There are now two laws of cosines. One gives you an angle based on the edge lengths, like in Euclidean space, and the other gives you the side lengths based on the angles. For the first law of cosines, given a triangle with edge lengths a, b, and c, we have $\cos(\angle C) = \frac{\cos(c) - \cos(a)\cos(b)}{\sin(a)\sin(b)}$. Using Taylor polynomials to a couple of terms, we can see that the spherical law of cosines approaches the Euclidean version in its limit. With $\sin(x) \approx x$ and $\cos(x) \approx 1 - \frac{x^2}{2}$ for x small, we get

$$
\begin{aligned}
\cos(\angle C) &= \frac{\cos(c) - \cos(a)\cos(b)}{\sin(a)\sin(b)} = \frac{(1 - \frac{c^2}{2}) - (1 - \frac{a^2}{2})(1 - \frac{b^2}{2})}{ab} \\
&= \frac{\frac{a^2}{2} + \frac{b^2}{2} - \frac{c^2}{2} + \frac{a^2 b^2}{4}}{ab} \\
&= \frac{a^2 + b^2 - c^2}{2ab} + \frac{ab}{4} \approx \frac{a^2 + b^2 - c^2}{2ab}.
\end{aligned}
$$

More importantly, our equation for combinatorial Ricci flow is altered slightly. The base equation, as given by [2], is now

$$
\frac{dr_i}{dt} = -K_i \sin(r_i). \tag{12}
$$

Like in the Euclidean case, we see the possibility that all weights could continually decrease towards zero. However, scaling in the spherical case is not quite as easy as before: the angles are affected by the change in edge lengths. Plus, we also need to make sure that the weights remain within bounds.

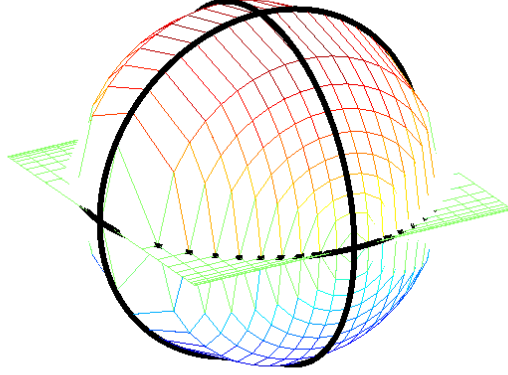We tried a few techniques to derive an equation for normalized spherical flow. Some ideas were:

Figure 16: An illustration of a spherical octahedron through the X-Y plane. Each octet of the sphere surface, outlined in black, is the face of one triangle.

- $\dfrac{dr_i}{dt} = (\overline{K} - K_i)\sin(r_i)$ By maintaining the same construction as the normalized Euclidean Ricci flow, we hoped that replacing r_i with $\sin(r_i)$ would work. However, in running even the simplest case of a tetrahedron, we found that the system was highly unstable. If more than one weight was initially different from the others, the program would fail. If the system was able to stabilize, we noted that the resultant surface area of the end product was 4π, the same as a unit sphere. Another issue with this formula is that the value $\overline{K}$ is no longer constant. For a spherical construction of a surface X, we have

$$\overline{K} = \frac{\sum K_i}{|V|} = \frac{2\pi\chi - \text{Surface Area of } X}{|V|}.$$

This is known as the Gauss-Bonnet theorem and is noted by Chow and Luo [2]. Since the surface area went to 4π, then the average curvature went to zero.

- $\dfrac{dr_i}{dt} = (\hat{K} - K_i)r_i$ Following the criteria we used to obtain our normalized Euclidean flow, we began experimenting and used the cases
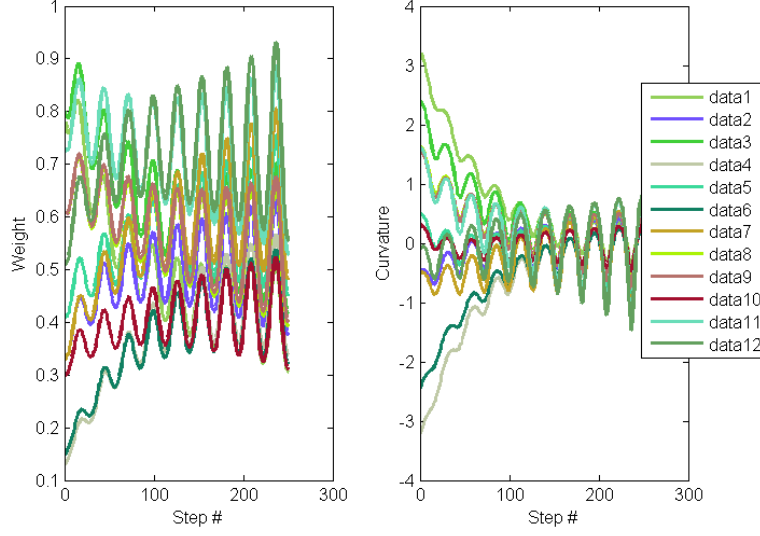
Figure 17: An example using our notion of $\hat{K}$. Weights and curvatures were unable to converge, and crashed the program shortly after.

$\tilde{r}_i = \alpha r_i$ and $\prod \alpha \sin r_i = C$ to obtain a result similar to our original. We defined $\hat{K}$ as the the dot product of the curvatures and the cosines of the weights, divided by the number of vertices, or

$$\hat{K} = \frac{\sum K_i \cos r_i}{|V|}.$$

The major problem we encountered with this formula was that we had to use a sine approximation in its derivation, which is not valid for larger weights. As angles are not conserved with scaling in the spherical case, we realized this formula would not work. We also found that weights and curvatures displayed sinusoidal behavior, but were unable to converge to a finite value. As time went on, the weights became more unstable until the program reached failure, as seen in Fig. 17.

In the end, we decided to try an equation that took the good aspects of our first trial while broadening its range of stability as we had hoped with the

second one. We came up with the equation

$$\frac{dr_i}{dt} = \overline{K}r_i - K_i \sin(r_i). \tag{13}$$

This equation was derived by applying (7) to the standard spherical equation and recognizing that the average curvature component should likely not be the sine of the weight, but just the product with the weight itself. Unfortunately, we have no proof that this equation is truly related to the original flow. However, in examining its behavior with various triangulations, we found that this one works very well, and acted similarly to that of the Euclidean normalized flow. We find that spheres converge to zero curvature and weights first group together and then approach optimal weights. See Figure 18.

In dealing with vertex transitive triangulations, ones in which each vertex has the same degree, we noted that sometimes all weights would converge to a specific value that was dependent on the number of vertices, and sometimes all the weights would be close to this number but off by a small amount. In either case, the resulting surface area at the end turned out to be 4π. For a vertex transitive genus 0 surface, we found that the preferred weight that each vertex approached or reached was equal to

$$r_i = \frac{1}{2} \arccos(\frac{\cos(Z)}{1 - \cos(Z)}) \text{ where } Z = \frac{\pi(4 + F)}{3F}$$

and F is the number of faces of the given triangulation. For example, with a tetrahedron, $F = 4, Z = \frac{2\pi}{3}$, and $r_i = \frac{1}{2} \arccos(-\frac{1}{3}) \approx 0.9553$.

We found with this equation to fail if some weights got too large, and the computation time required to reach a stable equilibrium was lengthier than expected. Looking at Fig. 18, we see that with a dt size of 0.5, it took over 500 steps for the system to reach equilibrium, and this would be much longer with the dt used in §4.4.1.
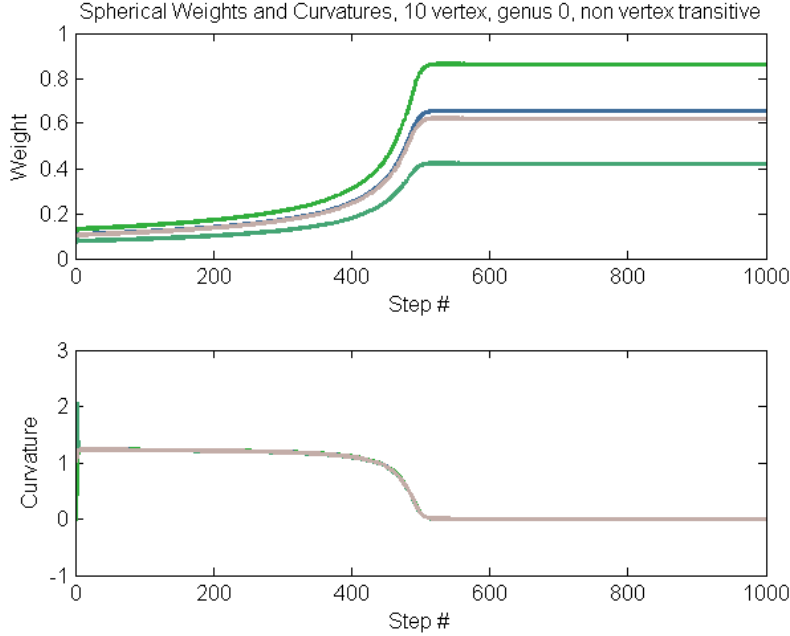
Figure 18: An example of a solution using Eq. (13). Starting off with equal initial weights of 0.1, vertices merge into one of four groups as they approach uniform curvature relatively quickly. As curvature drops to zero, slowly at first but more quickly as time passes, the weight groups separate from each other to obtain their final weight. In this trial, $dt = 0.50$ to show the complete process.

5.2 Hyperbolic flow

In hyperbolic geometry, we encounter a new background that has properties both similar and distinct from Euclidean and spherical systems. A common way to visualize the hyperbolic plane can be seen in Fig. 19. This representation is often called the Poincaré disk, named after the same Poincaré as in §4.

The hyperbolic plane has are several interesting properties. For a line l and a point P not on that line, there are an infinite number of lines through P that do not intersect l. In Euclidean geometry, such a line is unique. The shortest distance between two points is still a straight line, but illustratively can be

40

found by following a curved path along a circle centered at infinity going through both points. A more thorough exposition on hyperbolic geometry can be found in most college geometry textbooks.

In terms of the formulas, they remain relatively similar in format to the spherical equations, except there are two main substitutions. The cosine and sine functions are often replaced with their hyperbolic counterparts *cosh* and *sinh*, defined as

$$\begin{aligned}
\sinh(x) &= \frac{e^x - e^{-x}}{2} \\
\cosh(x) &= \frac{d\sinh(x)}{dx} = \frac{e^x + e^{-x}}{2}.
\end{aligned}$$

The equation for combinatorial Ricci flow is now

$$\frac{dr_i}{dt} = -K_i \sinh(r_i) \tag{14}$$

Like in the case of spherical flow, the average and total curvatures need not remain constant. The average curvature is now

$$\overline{K} = \frac{\sum K_i}{|V|} = \frac{2\pi\chi + \text{Surface Area of } X}{|V|}.$$

After running the hyperbolic Ricci flow on a few samples, we noted that the weights did not always go to zero, as was the case with the previous unnormalized systems. So in some cases, it almost seems as if this equation is already normalized. Curvatures would converge to zero, and optimum, nonzero weights are achieved in a relatively short time. Other times it behaved like both Euclidean and spherical. Weights would become chaotically large or drop to zero.

5.3   Comparison of Systems

When we began looking into these background geometries, we thought that it would be a good idea to run uniform tests on all three systems. We could observe whether or not each triangulation converges in the same fashion, or
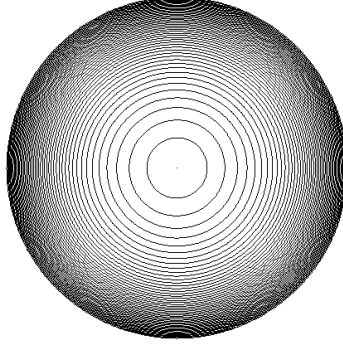
Figure 19: An illustration of the hyperbolic plane using the Poincaré disk. All concentric circles are evenly spaced apart in a hyperbolic background, but the outer edge of the disk approaches infinity, so the circles appear closer together.

if it matters which geometry we choose. If we do discover that all systems behave the same way, we would then like to focus on one geometry, most likely Euclidean. However, since our equation for normalized spherical Ricci flow is not wholly confirmed as the best method, and as the hyperbolic flow may or may not already be normalized, we can not make any strong conclusions on the normalized flows. We did decide, however, to take a look at the behaviors on various manifolds by using the unnormalized equations, (4), (12), and (14). We will look at three triangulations, each of a different genus, without performing any morphs, and compare results between each system.

In terms of programming these new flows, we were able to adapt our *calcFlow* program into two new programs, *sphericalCalcFlow* and *hyperbolicCalcFlow*, to easily distinguish which background we were using. This way we could run the systems one at a time, or all at once and observe differences in their behaviors in cases with the same initial conditions.

Example: 12 vertex sphere, vertex transitive of degree 5

Euclidean- As expected, we obtain a surface where all the vertices attain the same curvature of $\frac{\pi}{3}$ after multiple trials of uniform and randomized weights. The weights appeared to group together as in the normalized spherical case, but they were all decreasing towards zero.

Spherical- We found that the system was truly stable with nonzero final weights if and only if all initial weights were set exactly to its preferred weight, which is ≈ 0.55357 based on the fact that this triangulation has 20 faces. If the initial curvature was below zero, the weights would increase rapidly until the program crashed. If the initial curvature was positive, the vertices would approach the same curvature as in the Euclidean case. In doing so, the weights would drop to zero.

Hyperbolic- We found that the vertices react in a similar manner to the Euclidean case. They approach the same curvature of $\frac{\pi}{3}$ and drop towards zero weights at the same rate.

Example: 9 vertex, one-holed torus

Euclidean- Since we know that $\chi = 0$ for a one-holed torus, we also know that $\overline{K} = 0$ and so each vertex obtains zero curvature. The weights do not drop to zero, but converge to positive values reflective of their degree.

Spherical- We found this system to be very unstable with the torus. If the initial weights were too large, we found that the total curvature would be well below zero, and the weights would continually increase until spherical-CalcFlow crashed. While we hoped that reducing the initial weights would bring stability to the system, we ended up with the same result.

Hyperbolic- While the total curvature of the system goes to zero, the weights also approach zero but maintain roughly the same proportion as the final Euclidean weights. Another thing we noted was the time required for the weights and curvatures to go to zero. This system was extremely slow and took much longer than either the Euclidean or spherical trials.

Example: 11 vertex, two-holed torus

Euclidean- As in previous cases, the vertices converge to a uniform curvature. However, as this value was negative (given $\chi = -2$) we saw the weights increase without bound. As there is no limit to the weights in the Euclidean case, *calcFlow* had no troubles calculating each iteration, but having unbounded weights is still an undesired result.

Spherical- Regardless of the initial weights, we found that the total curvature of the system would continually decrease, and as such the weights of each vertex would increase until sphericalCalcFlow crashed.

Hyperbolic- Here we found that hyperbolic geometry was very effective. In a very short time, we saw the vertices reach zero curvature, and the weights converge to nonzero values. This is how we got our notion that the hyperbolic Ricci flow was already normalized. This also coincides with the suggestion in [2]; when working with a negative Euler characteristic manifold, one should use hyperbolic geometry.

To conclude, we conjecture that certain flows work best for triangulations whose total curvature goes to zero in that particular system.

- Euclidean is best for systems of genus 1, or one-holed tori. As seen in the second example, the unnormalized system converged weights to nonzero values.

- Spherical combinatorial Ricci flow should work best on manifolds of genus 0, or anything topologically equivalent to a sphere. This is because we have a positive χ value, so for total curvature to go to 0, the total surface area must go to 4π. Spherical was the only system that could have produced a nontrivial solution in our first example.

- Hyperbolic Ricci flow is best suited for triangulations with a genus of 2 or more. We can see that these systems will be able to reach zero total curvature with a large enough surface area.

6 Future work

6.1 Linking Delaunay to Ricci flow

Delaunay surfaces and combinatorial Ricci flows can seem to be very separate concepts, and in many ways with our program structure, they are. Yet there is the possibility of combining both of them to explore other mathematical inquiries. For instance, how does the idea of Delaunay triangles interact with triangulations on a manifold. We know that a circle packed triangulation is weighted Delaunay, but what if the triangulation is not circle packed (see below)? Will the Ricci flow lead to a weighted Delaunay triangulation?
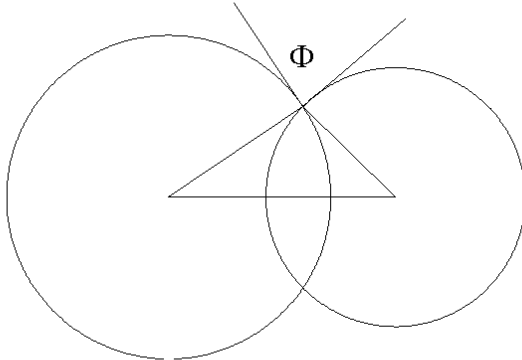
Figure 20: An example of relaxing circle packing and introducing Φ.

6.2 Circle packing expansions

Circle packing is but a very special way to characterize our side lengths. If we relax this criteria and let the circles overlap, we can introduce a second weight Φ that is representative of the angle of their intersection. See Figure 20. If we let Φ be constant, we can then evaluate our side lengths as

$$l_{ij} = \sqrt{r_i^2 + r_j^2 + 2r_i r_j \cos(\Phi(e_{ij}))}$$

With this more general interpretation, we can examine questions asked by Chow and Luo in [2].

6.3 Spherical and hyperbolic extensions

While we investigated various tests to compare the three combinatorial Ricci flows, we would like to investigate spherical and hyperbolic even further. Primarily, we would like to determine if our normalized equation, Eq.(13) is related to the Euclidean normalized flow or determine if a true normalized flow exists. Likewise, we would like to investigate the existence of a normalized hyperbolic flow that behaves like the Euclidean version. We would also

like to implement a scaling notion per vertex in the spherical case so that the weights could increase without necessarily crashing the program.

6.4 3-Dimensional triangulations

Triangulations exist outside of two dimensions and one future goal is to perform flows in a three dimensional setting. The flow in this case is known as Yamabe flow, discussed in [3]. While Yamabe flow is similar to Ricci flow, its value of K_i is determined quite differently, involving not only the angles of the faces, but also the cone angles associated with tetrahedron vertices. The structure is in place from this project to accomplish this goal. To help the understanding of both the two and three dimensional triangulations, we would like to create a way to visualize them and explore the surfaces locally.

7 Conclusion

There were many things we wanted to explore with triangulations, particularly towards Delaunay surfaces and combinatorial Ricci flow. A major aspect of our research was the creation of a functional and adaptable program to provide meaningful data. We feel that we have accomplished this goal. We have added a user interface to the program (see §A.6) for a straightforward way to run flips or flows. We hope that in the future this program can be used by others to perform their own tests.

Our work on Delaunay surfaces is unfinished, but we are excited about the prospects introduced by negative triangles. The research performed in this paper is the beginning of what we hope is a development of these negative triangles and of universally accepted definitions about them. In combinatorial Ricci flow, we have helped confirm the assertions given by [2] with numerical results in the Euclidean case. We have seen how certain properties of a weighted triangulation behave under combinatorial Ricci flow such as the weights and curvatures. Under Spherical geometry we remain uncertain as to the validity of our normalized Ricci flow, but are optimistic that such a formula does exist. Along with the above mentioned areas of further work, we also propose research in 3-dimensional Yamabe flow and in Delaunay algorithms on manifolds.

In addition to the clearly defined goals in the paper, we were able to experience a large scale project that often required independent work. We also became knowledgeable in an area of mathematics that is still young in its development. For these things we would like to thank Dr. David Glickenstein for being our mentor and guide on this project, Dr. Robert Indik and the University of Arizona Math Department for their help and support, and the National Science Foundation VIGRE #0602173.

References

[1] M. Brown. *Ordinary Differential Equations and Their Applications*. Springer-Verlag, New York, NY, 1983.

[2] B. Chow and F. Luo. *Combinatorial Ricci Flows on Surfaces*. Journal of Differential Geometry 63, Volume , 97-129, 2003.

[3] D. Glickenstein. *A combnatorial Yamabe flow in three dimensions*. Topology 44, 791-808, 2005.

[4] D. Glickenstein. Geometric triangulations and discrete laplacians on manifolds. Material given to us by David, 2008.

[5] F. H. Lutz. The manifold page. http://www.math.tu-berlin.de/diskregeom/stellar/.

[6] Dana Mackenzie. The Poincaré conjecture proved. http://www.sciencemag.org.

[7] J-P Moreau. Differential equations in c++. http://pagesperso-orange.fr/jean-pierre.moreau/c_eqdiff.html.

A Appendix

A.1 Derivation of Eq. (9)

We used the criterion

$$f(\tilde{r}_1, \tilde{r}_2, \ldots, \tilde{r}_n) = \prod \tilde{r}_i = \prod \alpha r_i = \alpha^n \prod r_i = C$$

to constrain the values of radii. We take the derivative of f with respect to t and obtain

$$
\begin{aligned}
\frac{df}{dt} &= n\alpha^{n-1}\frac{d\alpha}{dt}r_1 r_2 \ldots r_n + \alpha^n \frac{dr_1}{dt}r_2 r_3 \ldots r_n \\
&+ \alpha^n r_1 \frac{dr_2}{dt}r_3 r_4 \ldots r_n + \ldots + \alpha^n r_1 r_2 \ldots r_{n-1}\frac{dr_n}{dt}.
\end{aligned}
$$

But since $\dfrac{dr_i}{dt} = -K_i r_i$ from Eq. (4) we obtain

$$
\begin{aligned}
\frac{df}{dt} &= \frac{n\alpha^n}{\alpha}\frac{d\alpha}{dt}r_1 r_2 \ldots r_n - K_1 \alpha^n r_1 r_2 r_3 \ldots r_n \\
&- K_2 \alpha^n r_1 r_2 r_3 r_4 \ldots r_n - \ldots - K_n \alpha^n r_1 r_2 \ldots r_{n-1} r_n
\end{aligned}
$$

from which we can group terms and obtain

$$
\begin{aligned}
\frac{df}{dt} &= (\alpha^n r_1 r_2 \ldots r_n)(\frac{n}{\alpha}\frac{d\alpha}{dt} - K_1 - K_2 - \ldots - K_n) \\
&= C(\frac{n}{\alpha}\frac{d\alpha}{dt} - K_1 - K_2 - \ldots - K_n).
\end{aligned}
$$

If we assume the product is a constant, we have $\dfrac{df}{dt} = 0$. Thus we have

$$\frac{n}{\alpha}\frac{d\alpha}{dt} - K_1 - K_2 - \ldots - K_n = 0.$$

Rearranging we have

$$\frac{1}{\alpha}\frac{d\alpha}{dt} = \frac{d(\log \alpha)}{dt} = \frac{K_1 + K_2 + \ldots + K_n}{n} = \overline{K}$$
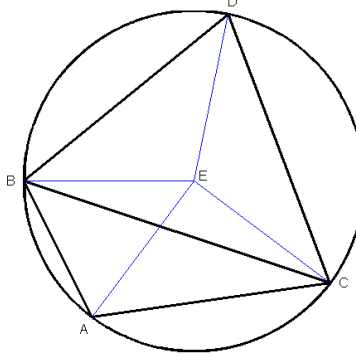
which we refer to as Eq. (9)

Figure 21: An illustration associated with our proof of Proposition 3.1.

A.2   Proof of Proposition 3.1

The sum of the opposite angles on a non-Delaunay hinge is greater than π.

Proof. Let ΔABC and ΔBCD be triangles with common edge $\overline{BC}$ (see fig. A.1). Let r be the radius of the circumcircle of ΔABC and E be the center of that circle. Then we wish to show that if $\overline{DE}$ is less than r, then $\angle BAC + \angle BDC > \pi$.

We know that $\overline{AB} = \overline{AE} = \overline{EC} = r$, so $\angle EBA = \angle BAE$ and $\angle EAC = \angle ACE$. Therefore, $\angle EBA + \angle ACE = \angle BAC$.

We first consider $\overline{DE} = r$. Then similar to above, $\angle EBD = \angle BDE$ and $\angle EDC = \angle DCE$, so $\angle EBD + \angle DCE = \angle BDC$. Thus, $\angle BAC + \angle EBA + \angle ACE + \angle BDC + \angle EBD + \angle DCE = 2\angle BAC + 2\angle BDC = 2\pi$ since $ABCD$ is a quadrilateral. So $\angle BAC + \angle BDC = \pi$.

Now we suppose $\overline{DE}$ is less than r. So $\angle BDE > \angle EBD$ and $\angle DCE > \angle EDC$. Therefore we have

$$\angle BAC + \angle EBA + \angle ACE + \angle BDC + \angle EBD + \angle DCE = 2\pi$$
$$\Rightarrow 2\angle BAC + 2\angle BDC > 2\pi \quad \Rightarrow \angle BAC + \angle BDC > \pi.$$

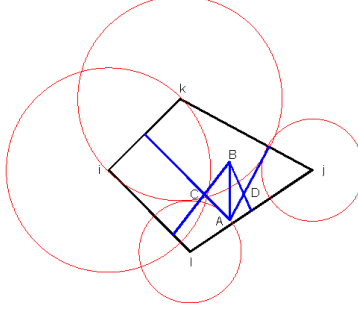Hence the sum of the opposite angles on a non-Delaunay hinge is greater

Figure 22: An illustration associated with our proof of Proposition 3.1.

than π. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

A.3 Proof of Proposition 3.4

Any hinge that is non-weighted-Delaunay will become weighted Delaunay after a 2-2 flip is performed on it.

Proof. Consider the hinge consisting of edge $\{i, j\}$ and faces $\{i, j, k\}$ and $\{i, j, l\}$. Assume this hinge is non-weighted-Delaunay. Let $A = C(\{i, j, k\})$ and $B = C(\{i, j, l\})$. We know that B must lie closer to k than A. Similarly, A is closer to l than B (see fig. A.2).

By definition of $C(\{i, j, k\})$, we can say that A lies at the intersection of $h_{ij,k}$, $h_{ik,j}$, and $h_{jk,i}$. Since it suffices to look at only two of these lengths, we will consider just $h_{ik,j}$ and $h_{jk,i}$. Similarly, we can say that B lies at the intersection on $h_{il,j}$ and $h_{jl,i}$. Because this hinge is weighted non-Delaunay, we know that there exists a point where $h_{ik,j}$ and $h_{il,j}$ intersect; the same can be said for $h_{jk,i}$, and $h_{jl,i}$. Let us call these points C and D respectively.

At this point, consider our hinge after a 2-2 flip has been performed on it. We now have faces $\{i, k, l\}$ and $\{j, k, l\}$. Since C is the intersection of $h_{ik,j}$ and $h_{il,j}$, we know that it also must be the intersection of $h_{ik,l}$ and $h_{il,k}$, which is $C(\{i, k, l\})$. Similarly, D must be $C(\{j, k, l\})$. We know that C must lie

51

closer to i than D and D must lie closer to j than C. This means that $\star\{k,l\}$ must be positive. Therefore, the hinge on $\{k,l\}$ is weighted Delaunay. $\quad\square$

A.4 Remarks on Runge-Kutta method for solving Eq. (11)

The method used by Moreau in [7] to solve a differential equation involves using a Runge-Kutta method. Prior to adapting the code from Moreau's website, we reached the conclusion that a Runge-Kutta format would be most beneficial for this type of differential equation problem. Even though it is more computationally complex than the simpler Euler's method, it makes up in its ability to converge and in its accuracy. According to [1] the error associated with using Runge-Kutta is on the order of h^4, whereas with a standard Euler approximation the error is simply of order h, with $h = dt$ being our step incremental.

Based on our evaluations of radii and curvatures over time, it appears to converge exponentially for each vertex. However, as mentioned previously, performing flips on a triangulation may create unusual behavior.

A.5 Data Plots

A.6 Code Examples

Code from the program can be found here: http://code.google.com/p/geocam.

The *calcFlow* method calculates the combinatorial Ricci flow of the current Triangulation using the Runge-Kutta method. Results from the steps are written into vectors of doubles provided. The parameters are:

- **vector<double>* weights-** A vector of doubles to append the results of weights, grouped by step, with a total size of numSteps * numVertices.

- **vector<double>* curvatures-** A vector of doubles to append the results of curvatures, grouped by step, with a total size of numSteps * numVertices. The time step size. Initial and ending times not needed since diff. equations are independent of time.
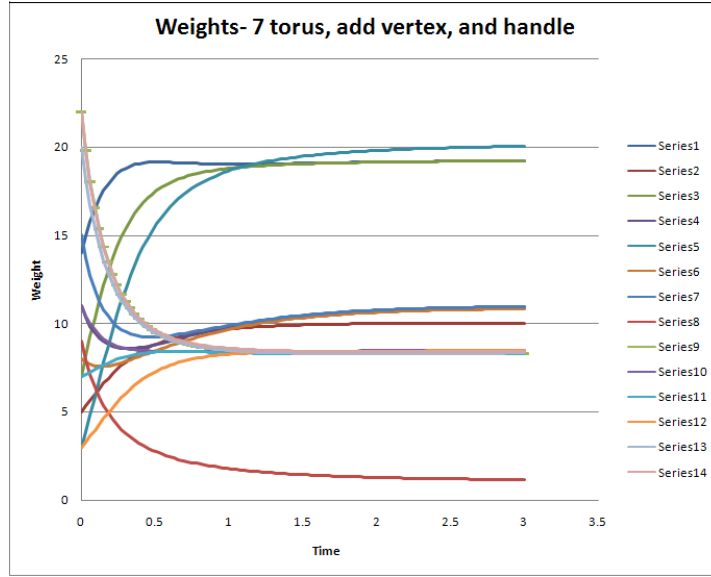
Figure 23: An example of how morphs can change the asymptotic behavior of vertices. In this case we saw the weights of some vertices change concavity.

- **double* initWeights-** Array of initial weights of the Vertices in order.

- **int numSteps-** The number of steps to take. (dt = (tf - ti)/numSteps)

- **bool adjF-** Boolean of whether or not to use adjusted differential equation. True to use adjusted.

The information placed in the vectors are the weights and curvatures for each Vertex at each step point. The data is grouped by steps, so the first vertex of the first step is the beginning element. After n doubles are placed, for an n-vertex triangulation, the first vertex of the next step follows. If the vectors passed in are not empty, the data is added to the end of the vector and the original information is not cleared.

```
void calcFlow(vector<double>* weights, vector<double>* curvatures, double dt,
            double* initWeights, int numSteps, bool adjF)
{
  int p = Triangulation::vertexTable.size(); // The number of vertices or
```
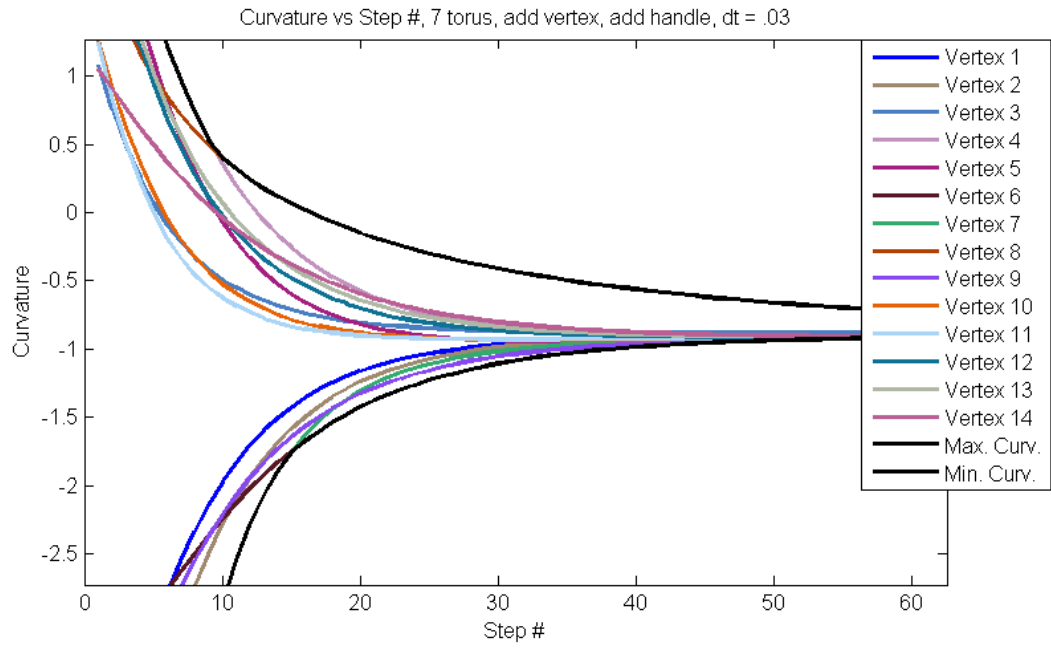
Figure 24: An example of curvatures over time. While they do converge to the same curvature, the vertex with the maximum or minimum curvature may change. This is a separate trial than that producing Fig. 23
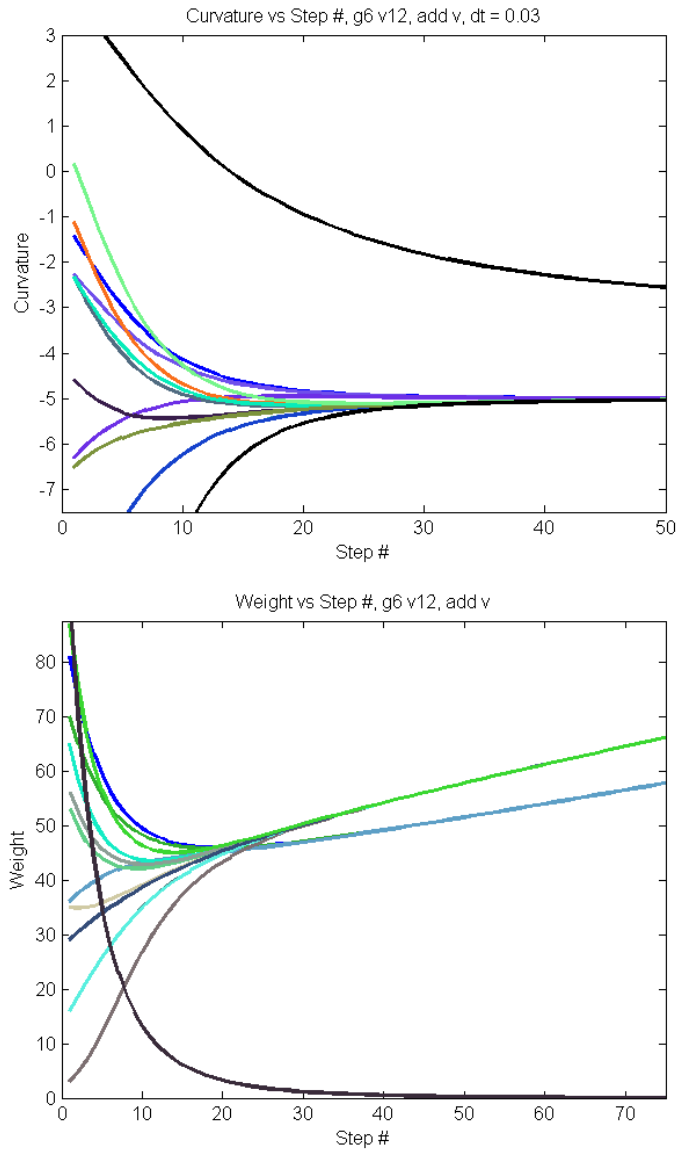
Figure 25: An example of adding a vertex to a genus 6 surface. One of the curvatures in unable to drop below $-\pi$, and as a result, its weight is pushed to almost zero. Other vertices group together to compensate for this behavior.
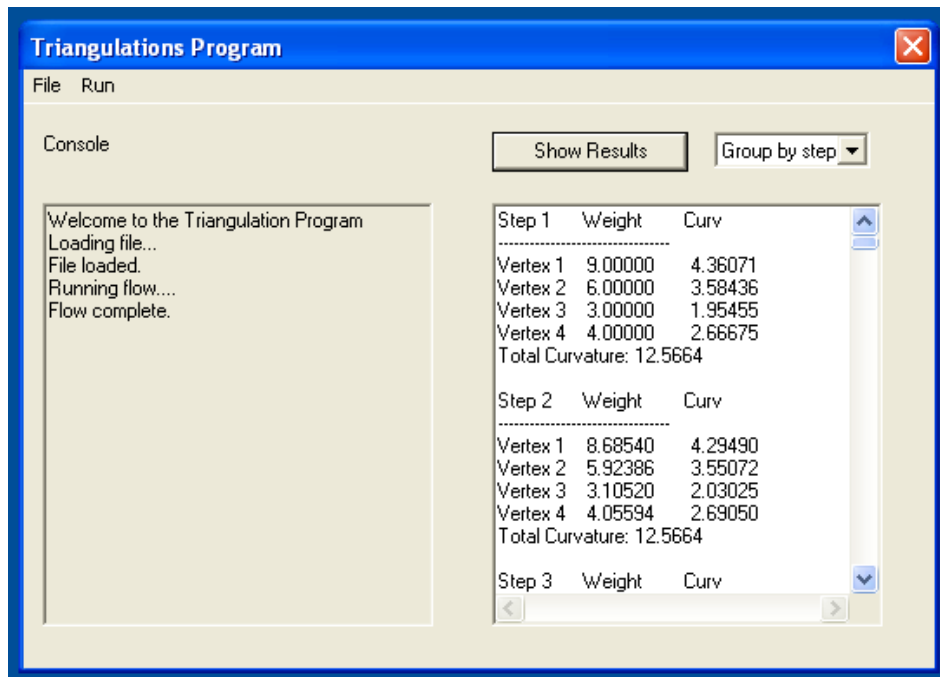
Figure 26: A snapshot of the user interface written to run both Ricci flows and Delaunay flip algorithms. Results of a flow are written in a text field on the right.

```
                                        // number of variables in system.

double ta[p],tb[p],tc[p],td[p],z[p]; // Temporary arrays to hold data for
                                    // the intermediate steps in.
int    i,k; // ints used for "for loops". i is the step number,
           // k is the kth vertex for the arrays.
map<int, Vertex>::iterator vit; // Iterator to traverse through the vertices.
// Beginning and Ending pointers. (Used for "for" loops)
map<int, Vertex>::iterator vBegin = Triangulation::vertexTable.begin();
map<int, Vertex>::iterator vEnd = Triangulation::vertexTable.end();

double net = 0; // Net and prev hold the current and previous
double prev;    //  net curvatures, respectively.
 for (k=0; k<p; k++) {
  z[k]=initWeights[k]; // z[k] holds the current weights.
 }
 for (i=1; i<numSteps+1; i++)
 {  // This is the main loop through each step.
  prev = net; // Set prev to net.
  net = 0;    // Reset net.

     for (k=0, vit = vBegin; k<p && vit != vEnd; k++, vit++)
     {
         // Set the weights of the Triangulation.
         vit->second.setWeight(z[k]);
     }
     if(i == 1) // If first time through, use static method to
     {          // calculate total curvature.
          prev = Triangulation::netCurvature();
     }
     for (k=0, vit = vBegin; k<p && vit != vEnd; k++, vit++)
     {  // First "for loop"in whole step calculates everything manually.
         (*weights).push_back( z[k]); // Adds the data to the vector.
         double curv = curvature(vit->second);
         if(curv < 0.00005 && curv > -0.00005)
         {  // Adjusted for small numbers. We want it to print nicely.
           (*curvatures).push_back(0.); // Adds the data to the vector.
         }
```

```
    else {
        (*curvatures).push_back(curv);
    }
    net += curv; // Calculating the net curvature.
    // Calculates the differential equation, either normalized or
    // standard.
    if(adjF) ta[k]= dt * ((-1) * curv
                    * vit->second.getWeight() +
                    prev /  p
                    * vit->second.getWeight());
    else    ta[k] = dt * (-1) * curv
                    * vit->second.getWeight();

}
for (k=0, vit = vBegin; k<p && vit != vEnd; k++, vit++)
{ // Set the new weights to our triangulation.
    vit->second.setWeight(z[k]+ta[k]/2);
}
for (k=0, vit = vBegin; k<p && vit != vEnd; k++, vit++)
{
    // Again calculates the differential equation, but we
    // still need the data in ta[] so we use tb[] now.
    if(adjF) tb[k]=dt*adjDiffEQ(vit->first, net);
    else     tb[k]=dt*stdDiffEQ(vit->first);
}
for (k=0, vit = vBegin; k<p && vit != vEnd; k++, vit++)
{ // Set the new weights.
    vit->second.setWeight(z[k]+tb[k]/2);
}
for (k=0, vit = vBegin; k<p && vit != vEnd; k++, vit++)
{
    if(adjF) tc[k]=dt*adjDiffEQ(vit->first, net);
    else     tc[k]=dt*stdDiffEQ(vit->first);
}
for (k=0, vit = vBegin; k<p && vit != vEnd; k++, vit++)
{ // Set the new weights.
    vit->second.setWeight(z[k]+tc[k]);
}
```

```
    for (k=0, vit = vBegin; k<p && vit != vEnd; k++, vit++)
    {
        if(adjF) td[k]=dt*adjDiffEQ(vit->first, net);
        else     td[k]=dt*stdDiffEQ(vit->first);
    }
    for (k=0; k<p; k++) // Adjust z[k] according to algorithm.
    {
      z[k]=z[k]+(ta[k]+2*tb[k]+2*tc[k]+td[k])/6;

    }
  }
}
```

About the authors

Alex Henniges is a junior double majoring in Math and Computer Science. Thomas Williams is a senior in Comprehensive Mathematics with a minor in Computer Science and a background in Math Education. Mitch Wilson is a senior double majoring in Applied Math and Mechanical Engineering.

Contact information:

- Dr. David Glickenstein- glickenstein@math.arizona.edu

- Alex Henniges- henniges@email.arizona.edu

- Thomas Williams- thwillia@email.arizona.edu

- Mitch Wilson- mjw@email.arizona.edu