Geometric Evolution on Computationally Abstract Manifolds

Alex Henniges
Thomas Williams
Mitch Wilson

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

July 2, 2008

Table of Contents

1 Introduction

The purpose of this project is to provide a method for examining the effect of combinatorial Ricci flow over various compact manifolds in two and three dimensions. The execution of this flow is made possible by the use of triangulations and the simulation of them inside a computational data structure. The design of this data structure will be integral in the development of this project. The program will be written in C++. Educationally, we hope to learn a number of things, including advanced topics in geometry and topology, programming in a new and unfamiliar language, and writing a professional report of our findings.

In this paper we will discuss the number of things we learned and worked on, beginning with an introduction to triangulations and their properties. Then we will establish the definition of Ricci flow and its use in our research. That will be followed by a discussion of how the program was structured as well as an explanation of several functions that were created. After this we will be ready to reveal the results and analysis we have obtained up to this point. Lastly, we will provide areas of future work in combinatorial Ricci flow and for this project.

2 Triangulations

2.1 Introduction

Suppose you are asked to make a sphere with as few pieces as possible. You could make a number of possible shapes, like a cube, or even a soccer ball. While both of these shapes are discrete in nature, they can be used to represent a continuous sphere. In fact, you are probably already familiar with the most basic representation of a sphere- the tetrahedron. Using only four vertices, six edges, and four faces, the tetrahedron is able to give us an approximation to a sphere, like how three vertices- a triangle- can be used to approximate a circle in two dimensions. Naturally, if we add more vertices, we are able to better illustrate our shapes. A tetrahedron can look like a soccer ball; a triangle can look like an octagon. Like in modern video games and Hollywood movies, we can generate various shapes of many different sizes using polygons that mold to form the special effect. For these and all

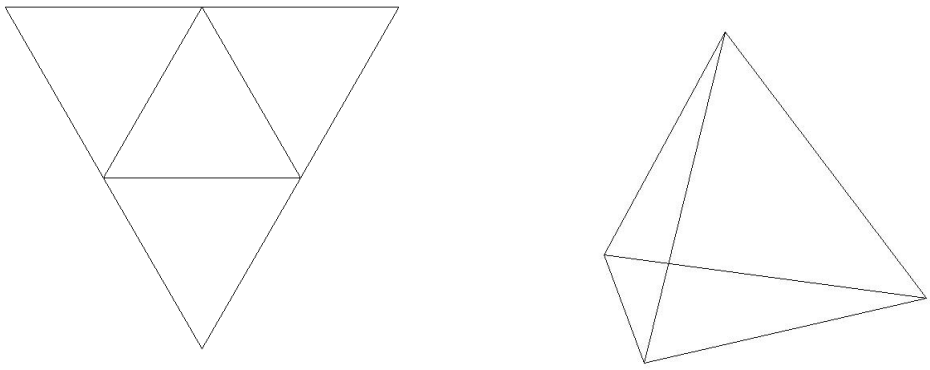

Figure 1: An example of triangulation. A triangle can be folded up into a tetrahedron.

shapes, we will focus on building them solely out of triangles. Since any regular polygon can be broken up into smaller triangles, we can essentially represent any shape with enough vertices. We define a shape as *triangulable* if we can connect all triangles in a particular fashion such that they create a closed 3-dimensional shape.

Of course, not everything is made of spheres, and while to the topologist, a soccer ball and a football and even a hockey puck are considered the same, spheres are not the only possible configuration we care about. Another shape of interest is known as the torus. It is the first object that is fundamentally different from the sphere. The reason is that there is a hole in the torus, which looks like a donut, or a bagel. It turns out that a visualization of this object can be made using as few as 7 vertices [5]. It is also possible to add "handles" to any shape, resulting in the addition of another hole, like adding a handle to a coffee mug. There are also various other shapes which can be represented with varying numbers of vertices, such as Klein bottles and cross-caps. [7] has more information on these fascinating shapes, and we will examine some of these further on.

An issue arises regarding the uniqueness of these triangulations. It is known that there are millions of ways to triangulate even the most basic shapes using a fixed number of vertices, but what separates spheres from tori is not the number of vertices used, but the number of holes that are created-

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

Table 1: Listings of vertices, edges, faces, and genus for some common shapes [9]

one of the few distinctions that can be made between two objects in the world of topology. The techincal term for this is the *genus* of an object. A sphere, or anything like a sphere, has genus 0. A torus, with its one hole, has genus 1, and a two-holed torus has genus 2. Another value associated with these surface is known as the Euler characteristic, χ. It is defined as $\chi = V - E + F$, where V, E, and F are the number of vertices, edges, and faces a given configuration has. For two-dimensional manifolds, it is also known that $\chi = 2 - 2g$, with g being the genus of the object. For example, any one hole torus has $\chi = 0$, and any representation of a sphere has $\chi = 2$. Table 1 has listings for many common shapes.

2.2 Definitions

A triangulation of an n-dimensional manifold is a way of representing the manifold as the composition of n-dimensional simplices. A simplex in n dimensions can be thought of as the simplest geometric representation of an object in n dimensions. For example, a 0-simplex would be a point, a 1-simplex is a line segment, a 2-simplex is a triangle, and a 3-simplex is a tetrahedron. For the purpose of this project, it will not be necessary to look at simplices of higher dimensions. However, concepts may remain to be represented in general terms.

By creating triangulations for these manifolds, we have created discrete structures to more easily run computations over. In a way, a triangulation can be thought of as a graph; an arrangement of points and the unique paths

between them. In order to run a flow over a structure like this, we look at the characteristics of each individual point. The development of this structure began by looking exclusively at 2-dimensional manifolds, also known as surfaces. This limits our structures to only three types of simplices, which are also known as vertices, edges, and faces.

As the triangulation of the manifold itself is defined by the composition of 2-dimensional simplices, these simplices are also defined by lower dimensional simplices. In order to create these structures, before giving any quantities to define things like edge length, we can define the structure by a series of lists of references. That is to say, every simplex is local to simplices of every other type. These lists of local simplices are what we use to define the triangulation itself. We say a vertex is local to any edge or face that it is in the definition of, as well as any vertex that it shares an edge and face with. Similarly, an edge is local to its defining vertices, and faces that it is in the definition of, and any edge that it shares a vertex with. We call this property adjacency. Thus, a vertex is adjacent to at least three other vertices, at least three edges, and at least three faces, an edge is adjacent to exactly two vertices and exactly two faces, and a face is adjacent to exactly three vertices, exactly three edges, and exactly three faces. The number of local edges that a vertex has is known as its degree.

2.3 Circle packing

A triangulation exists outside of lengths, but in order to perform any measurements we need a way to quantify our system. This is known as adding a metric to the triangulation, creating a *weighted triangulation*. This is done by a technique called 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 2. The radius of the circle at a vertex, v_i, is known as its weight, r_i. Beginning with triangles of pre-defined sizes, it can seem daunting and perhaps impossible to apply weights at each vertex to create a circle-packing. But working in the opposite order, applying weights to each vertex to define the size, is far easier. In this way, we provide lengths to the edges of our triangulation as $l_{ij} = r_i + r_j$. The metric applied to the triangulation in this way is known as the *cone metric* [2].

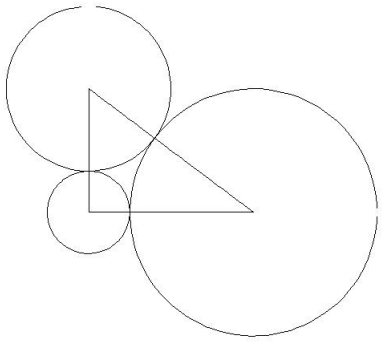

Figure 2: An example of circle packing. Three mutually tangent circles forming the perimeter of a triangle.

By applying weights first, we come across an interesting situation. Not all combinations of edge lengths are possible under this system. Some triangles can simply not be circle-packed. The most common example is shown in figure , where there is no combination of weights at the vertices that can make a proper circle-packing. By allowing more general circle-packings, as discussed in 6.2, we can broaden our range of possible weighted triangulations.

2.4 Duals

Every triangle that is capable of being circle-packed has a circle that is internally tangent to all three edges. It is known that this *incircle* is perpendicular to each edge and has a radius $r = \sqrt{\frac{(s-a)(s-b)(s-c)}{s}}$ where s is the semi-perimeter of the triangle, and $a, b,$and c are the side lengths. Let us rewrite this in terms of the vertex weights, since we are using circle packing. Thus, a side length is simply the sum of two vertex radii. We can then simplify this equation to $r = \sqrt{\frac{r_i r_j r_k}{r_i + r_j + r_k}}$, with $i, j$and k being the vertices of the triangle. This can be repeated for multiple triangles. All these incircles are known to be mutually tangent to each other, and any line connecting two adjacent incircles is perpendicular to the common edge of the two triangles. We define the length of this dual edge $\star e$ as the sum of the radii of adjacent

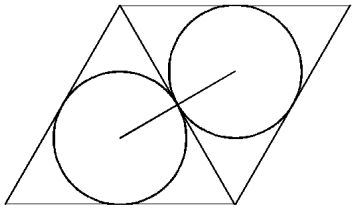

Figure 3: A dual to an edge

incircles. $\star e = r_{in1} + r_{in2}$. Every edge has its own dual length. See Figure 3 for an illustration of a dual edge.

With enough dual edges, we can obtain a polygon surrounding a vertex. The area of the region formed by these duals is called the dual area, evaluated by

$$\star A_i = r_i \sum \frac{\star e_i}{2} = r_i \sum r_{\text{inscribed}}$$

where r_i is the original weight of the vertex.

The dual length and dual area have some interesting properties that we hope to investigate. David Glickenstein examines these and more in [4].

3 Ricci flow

3.1 Background

Introduced by Richard Hamilton in 1982, Ricci flow, named in honor of Gregorio Ricci-Curbastro [6], has since made large impacts in the world of geometry and topology. It is often described as a heat equation. 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 even.

But the biggest consequence of Ricci flow was not to become apparent until early after the turn of the millennium, when Grigori Perelman presented three papers providing the proof to the Poincaré Conjecture. The Poincaré Conjecture, proposed in 1904, proved to be particularly difficult to prove, and was given the honor of one of seven millennium puzzles by the Clay Mathematics Institute. It was Ricci flow that turned out to be the cornerstone for the proof. In addition, its relation to the heat equation may open new doors for work in fluid dynamics and even in the theory of general relativity [6].

3.2 Definition

Just because we have a triangulation and that it is circle packed does not mean that it is already compact and organized. Vertices can be too large or too small, and the end shape can be somewhat repulsive. We would like to have a way to determine the evolution of each shape to its final form, which may be more uniform. We introduce to the reader the equation for combinatorial Ricci flow, which allows an individual radius to change over time on a discrete surface. In Euclidean space, the differential equation can be written as

$$\frac{dr_i}{dt} = -K_i r_i \tag{1}$$

where K_i is a characteristic called the curvature of a vertex, and r_i is the radius or weight of a vertex i. The value of K_i changes with time. Its value is found by determining the angles of all triangles containing vertex i. 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:

$$\angle C = \arccos \frac{a^2 + b^2 - c^2}{2ab}$$

with similar formulas for the other angles. We take the sum of all angles associated with a vertex i and define the curvature K_i as:

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

Since the differential equation depends on a variable whose value changes over time, we cannot solve it so easily.

A potential issue we noted is that based on the equation, more often than not, the lengths of the radii would decrease. Take the example of 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 by the same amount, but the curvature of each vertex would still remain π. The radii would continue to decrease until they approach zero length. We then have to address that issue since computers don't like working with numbers near zero, as in the denominator of the arccosine function. Weird, unwanted stuff might happen. Let us introduce the ability to resize the length of each radius by a scalar, α. We denote each scaled length by $\tilde{r_i}$ and equate as

$$\tilde{r_i} = \alpha r_i \tag{3}$$

Thus in plugging in to the differential equation we get

$$\begin{aligned} \frac{d\tilde{r_i}}{dt} &= \frac{d(\alpha r_i)}{dt} = \alpha \frac{dr_i}{dt} + r_i \frac{d\alpha}{dt} \\ &= -\alpha K_i r_i + \frac{\tilde{r_i}}{\alpha} \frac{d\alpha}{dt} \\ &= -\tilde{K_i} \tilde{r_i} + \frac{\tilde{r_i}}{\alpha} \frac{d\alpha}{dt} \end{aligned} \tag{4}$$

Since we are scaling all sides by the same factor, this does not effect the curvature of the surface, so $\tilde{K_i} = K_i$. It also turns out that

$$\frac{1}{\alpha} \frac{d\alpha}{dt} = \frac{d(\log \alpha)}{dt}$$

In order to find an appropriate value for α we decided to use the following criteria:

$$f(\tilde{r_1}, \tilde{r_2}, \ldots, \tilde{r_n}) = \prod \tilde{r_i} = \prod \alpha r_i = C, \text{ a constant} \tag{5}$$

This can be used to prevent radii from decreasing to zero. It turns out if you take the derivative of (5) with respect to time you find that

$$\frac{d(\log \alpha)}{dt} = \frac{\text{sum of all curvatures}}{\text{number of vertices}} = \overline{K}, \text{average curvature} \tag{6}$$

An in-depth proof of this can be found in the Appendix. It turns out that $\overline{K}$ is constant for any iteration of our triangulation and is equal to $\frac{2\pi\chi}{|V|}$, where $|V|$ is the number of vertices and χ is the Euler characteristic. This is noted in [2]. Plugging this back into (4) we determine that

$$\frac{d\tilde{r_i}}{dt} = -\tilde{K_i}\tilde{r_i} + \overline{K}\tilde{r_i} = (\overline{K} - K_i)\tilde{r_i} \tag{7}$$

However, since everything is now a function of $\tilde{r_i}$ and not α, we can treat α as a constant and thus $\frac{d\alpha}{dt} = 0$. Therefore, we can easily plug $\tilde{r_i} = \alpha r_i$ back in to the differential equation, so we end up with:

$$\begin{aligned} \frac{d\tilde{r_i}}{dt} &= (\overline{K} - K_i)\tilde{r_i} \\ \alpha\frac{dr_i}{dt} &= (\overline{K} - K_i)\alpha r_i \\ \frac{dr_i}{dt} &= (\overline{K} - K_i)r_i \end{aligned} \tag{8}$$

This is known as normalized Ricci flow, as discussed in [2]. In the case of our basic tetrahedron from earlier, the radii would not change after each iteration as $\overline{K} = K_i = \pi$ for each vertex and thus $\frac{dr_i}{dt} = 0$. With its roots in differential geometry, Ricci flow has become a popular topic in mathematics, finding uses and applications in many different fields.

3.3 Expectations

With regards to our program, Ricci flow over 2-manifold Euclidean surfaces will be important since a lot is already known about them. Cases like the tetrahedron can be calculated by hand so it will be useful to test our results against these surfaces. For example, we expect that the Tetrahedron under (1) will have all weights converge to zero. Whereas under (8) the weights are expected to converge to positive constants.

There are other expectations beyond the simple cases we can calculate by hand. According to [2], using the normalized Ricci flow will create a constant "area". This makes sense, as we fixed the product of the radii to be constant

in (5). Thus, we expect that the product of the initial weights will be equal to the product of the weights 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.

Out of our program we expect to create a system that allows for easy access of information while also providing that information in a time efficient way. 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 find them and fix them with proper testing.

4 Program/Code

4.1 Structure

When beginning the creation of the program, the structure design was critical. Not only would it help dictate the direction of the project over the course of its lifespan, but the design decisions would affect the speed and efficiency of all added functionality. It was agreed that the system would have to hold the different simplices and that they would be referencing each other. This part of the program would need to be structured in a way that was quick and easy to move from one simplex to another. As seen in figure 4, all simplices are assumed to have lists of references to other simplices, what we call local simplices, broken down by dimension. So for the two-dimensional case, each simplex has lists of local vertices, local edges, and local faces, as shown in Table 2.

The lists are vectors of integers. The vector, provided in the C++ library,

Vertices	Edges	Faces
adjacent vertices	component vertices	component vertices
constructed edges	adjacent edges	component edges
constructed faces	constructed faces	adjacent faces

Table 2: A layout of how data is organized

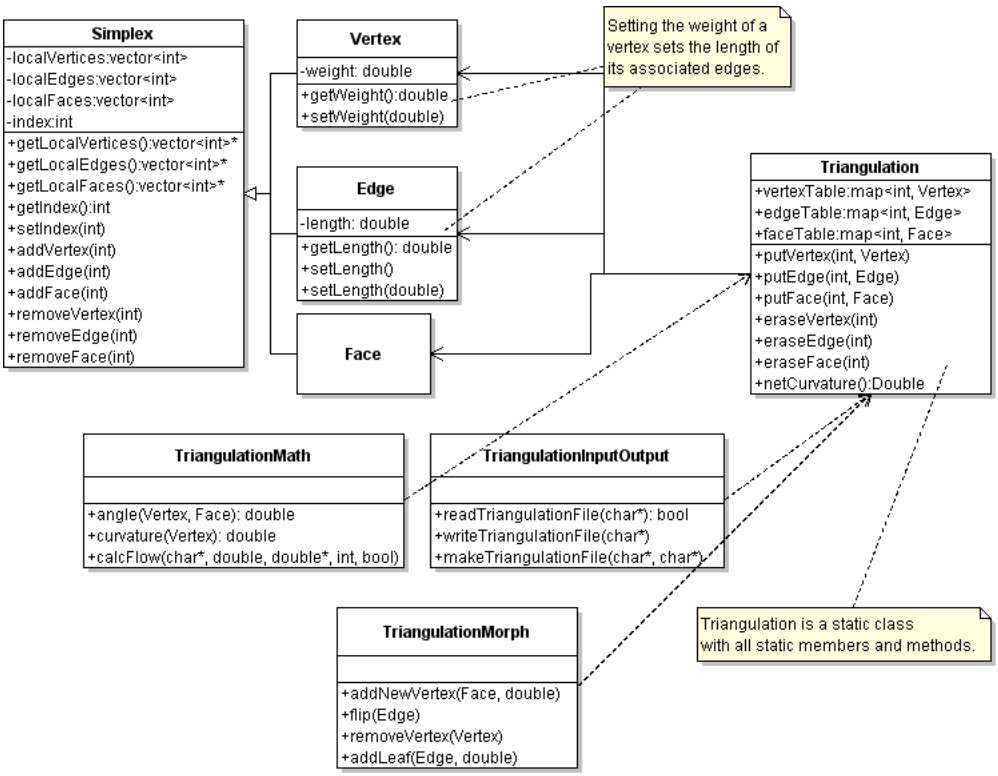

Figure 4: A UML diagram of the program.

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 is made up of static methods 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. Vertices also hold a weight, representing the radii from a circle packing on the triangulation. Edges then have a length based on the weights of its two vertices. When a vertex's weight is changed, the local edges to that vertex act as an observer and will update their lengths automatically.

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 3: Conversion of format from [5] to ours.

While we are able to manually construct a few basic triangulations by hand, as we add more vertices doing so will become ever harder. [5] has millions of known triangulations of varying size. 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 be able to 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, which can be seen in Table 3.

4.2 calcFlow

The function *calcFlow* runs a Ricci flow over the triangulation and records the data in a file. The algorithm for the ODE, provided by J-P Moreau, employs a Runge-Kutta method [8]. First, the file name for the data to be written in 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

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
Net Curv: 0.0000			Net Curv: 0.0000		

Table 4: Two steps of a Ricci flow

record. Lastly, a boolean is provided, where *true* indicates that the adjusted differential equation that is scaled to a constant, equation (8), should be used. Otherwise, the standard equation (1) is employed. Printed to the file is each step with every vertex and its weight and curvature at that moment. At the end of each step, the net curvature is printed, as 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 net curvature, which remains constant in two-dimensional 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 8.3

4.3 Morphs

Another part of the code is a file strictly made up of functions used to manipulate and alter the triangulations being run. 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.

4.3.1 Flips

One type of morph that we have available is known as a flip, called this because the transformation appears equivalent to the change in the planar projection of a tetrahedron when it is flipped over completely. Known also as "bistellar moves", these transformations can be used to change the degree of a particular vertex or many vertices, or coarsen or refine an entire triangulation as a whole. In higher dimensions, there are many different moves of this kind, but in two dimensions, we have only three unique types.

- 1-3 flip
  In a 1-3 flip, a single face is replaced by three faces. One way to think of going about this procedure is to simply add a new vertex to an existing face. This requires creating three new edges that connect the new vertex to the three defining vertices of the existing face.

- 2-2 flip
  In this flip, we take a pair of adjacent faces and readjust the edge that they share to connect the opposite pair of vertices, as shown in figure 5. This move creates no new simplices and removes no existing vertices. The code for the 2-2 flip is located in 8.3.

- 3-1 flip
  In what is essentially the inverse of the 1-3 flip, the 3-1 flip is one where we take a vertex that is connected to three other vertices and remove it entirely. This move is more restrictive than the others since not every vertex is connected to exactly three vertices.

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 $(V+1)-(E+3)+(F+2) = V-E+F = \chi$.

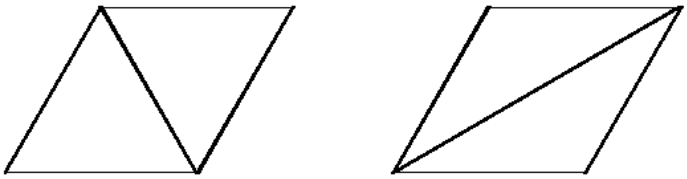

Figure 5: An example of a 2-2 flip.

This means we could do an infinite number of these flips without changing the topology of the surface.

Another thing of note is that any flip we could think of (2-5 flip, 1-4 flip) could be made using a combination of 1-3, 2-2, and 3-1 flips. Thus these three act as a basis for the span of any possible legal configurations.

4.3.2 Other Transfigurations

In addition to the flips that were described above, there are other "morphs" that can be performed to change the actual 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. In particular, there are only 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 of obtaining this result is to "patch" the surface of a torus to the surface that we are working with. With our code, this is exactly what we have done. The method that we have written removes a face from the existing triangulation and replaces it with a 9-vertex triangulation of a torus, the standard flat model. 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 net curvature of the figure.

- Adding Cross-Caps
  A cross-cap is a piece of a surface with a self intersection. 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. Another way to describe this is to say that the surface has only one side. In this way, the portion of the surface known as the cross-cap is, itself, topologically equivalent to the mobius band. Since this property of non-orientability is one without any level or degree, it remains sufficient to add only a single cross-cap to any given surface, but in conjunction with the method of adding handles, this technique allows us to create any topology within two dimensions.

- Adding Double Triangles
  Another shape we have been looking into is called the double triangle. In essence, a double triangle is made of two triangles with the same three vertices. It can also be thought of as folding one triangle on top of another. We can add these to a triangulation and see how things vary with their presence. This is of interest because of their very peculiar effect on the definition of a triangulation. Without double triangles, there are certain restrictions on definitions and adjacency properties of simplices within a triangulation as outlined in the previous section on triangulations. However, with these double triangles, the restrictions are broken up by allowing double and even triple adjacency references, creating vertices with only one or two local vertices, faces with only two defining edges, and edges with only one defining vertex, among other bizarre properties. Because of these inconsistencies in definition, different surfaces that have these double triangles attached to them will behave in unique and noteworthy ways. The function that adds these shapes takes an edge as an argument and "opens" it by creating another edge between the same two vertices, and then adds two more edges from these two vertices, meeting at a single new vertex and defining two new faces.

5 Results

5.1 Flows

The program for the Ricci flow was tested by beginning with the most simple cases, and then explored with as many different possibilities as we could conceive of to try to find anomalies. The first test was the tetrahedron. In the standard equation, it is easily shown that all weights approach zero exponentially fast. In the normalized equation, and the equation that is used in the rest of the testing, the tetrahedron's weights approach a single positive number, the fourth root of the "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 net 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 a constant multiple of π that was expected and the "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 extremes amongst the initial weights 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 feel fairly certain that there might be a formula relating the 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. The only explanation we could find with some merit is that 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

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

never less than four, while vertices 4 and 6 each have a local vertex with degree 3. See Figure 10 in the Appendix for an illustration of weights over time.

5.2 Specific cases

Example: Adding a vertex to a one-holed torus. See figure 6.

By inserting a vertex within a flat triangulation for a torus, we are essentially creating a bump on the torus and then observing what happens as we run it trough out *calcFlow* program. We discovered that 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. (5). The weight that the new vertex converges to turned out to be in proportion to the other weights by $3 + 2\sqrt{3}$, the exact proportion necessary to maintain equality amongst all other 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.

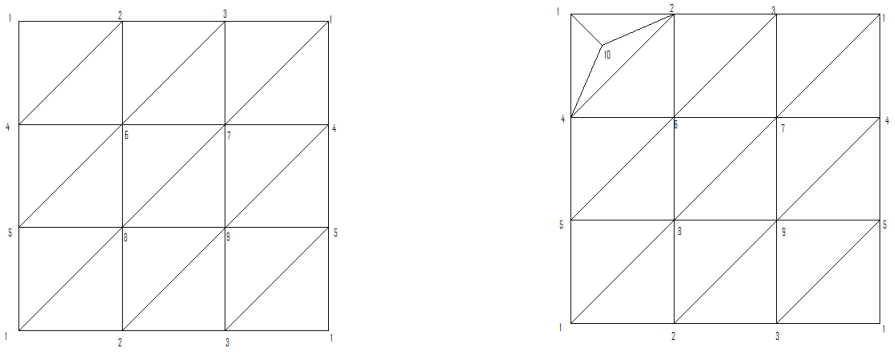

Figure 6: A triangulation of the torus, and the addition of a new vertex. This is a 1-3 flip.

Example: Adding a leaf 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 $\approx \pi$ 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.

One interesting observation we made was that flips can drastically change the behavior of some triangulations. For Example, in a 12-vertex torus, performing a flip on one edge affected 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.

Example: Triangulation of genus 4.

While the previous two cases were quite interesting, there is some hesitation given that we were using double triangles and being less restrictive in what were allowable triangulations. But the theory behind why both situations reacted as they did left hope for a case that fit in a stricter setting. In the same sense that a two degree vertex could have a curvature no less than 0, a three degree vertex is bounded below by $-\pi$. It was also clear that the vertices of a triangulation all converge to $\frac{2\pi\chi}{|V|}$. Now it was a matter of finding a triangulation with a large enough genus and few vertices. 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 try to converge to $-\pi$. We then expected that the new vertex would react as in the previous example. This was in fact the case, and it presents the question of whether or not there is a limit on the degree that can create such a singularity. The issue being that for higher genus, more vertices are required. Unfortunately, [5] does not provide large enough triangulations for fourth degree vertices.

Example: two tetrahedra connected at a vertex. See Figure 7

It turns out χ =7 Vertices - 12 Edges + 8 Faces = 3, which is not a case we had seen before. In previous cases χ was an even integer. Starting each vertex with equal weight, we obtained an 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. One good thing we noted was that the net curvature = $6\pi = 2\pi\chi$, which was a relief.

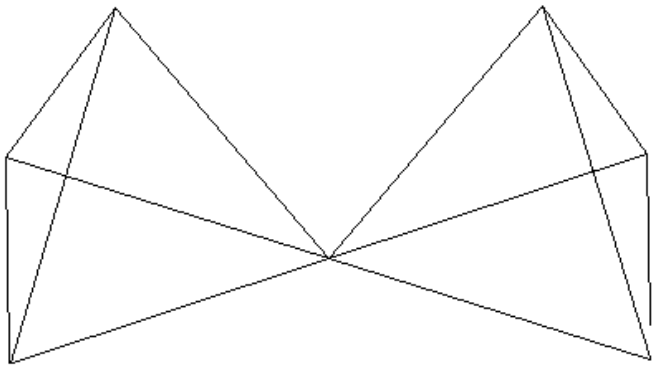

Figure 7: Two tetrahedra conjoined at a vertex.

5.3 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 only viable as it was unclear what set weights could be equal for all triangulations. The dt was held constant at 0.03. 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). The data collected at times seemed interesting and followed a pattern while other times were inexplicable.

Beginning with the basic case, the tetrahedron took on average 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

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 8: Summary of convergence data for varying criteria

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 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 unique vertices. By unique we mean the properties of the vertex (number of local vertices, convergence weight, 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.

6 Future work

6.1 3-D

We would also like to start investigating 3-dimensional constructs built from tetrahedrons. We can adjust our current code as much as we need to, and ultimately be able to evaluate Yamabe flow, as discussed by [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 internal angles associated with them.

Ultimately, Dr. Glickenstein would like to be able to build 3-D representations of these surfaces and walk along them in any given path. We can look into how faces are connected to each other, and like a tesselation, be able to move from one triangle to another seemlessly. We can lay groundwork for that, and when this project is able to jump to 3-D modeling, we hope that

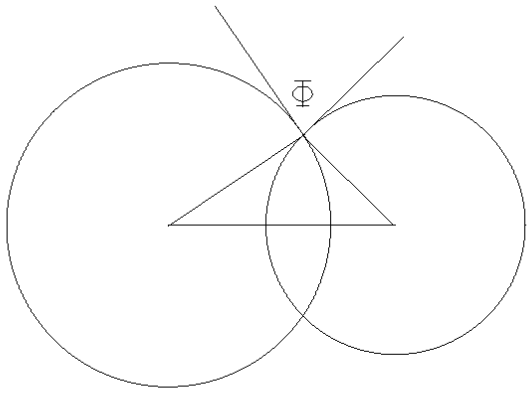

Figure 9: An example of relaxing circle packing and introducing Φ.

this will help.

6.2 Circle packing expansions

Circle packing is 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 how much they intersect by. See Figure 9. 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 Hyperbolic/ spherical triangulations

All this time we have focused on Euclidean coordinates. However, there are cases when triangulations may be better suited in other systems. In changing our format to hyperbolic and spherical coordinates, we would have to reevaluate each of our steps depending on the system. For example, in hyperbolic, the basic Ricci flow equation is $\frac{dr_i}{dt} = -K_i \sinh r_i$. Likewise, in spherical

coordinates, for a surface $X, \sum K_i = 2\pi\chi -$ Surface Area of$X$, which can change over time.

Another thing of interest is that once we run variations of our *calcFlow* program based on our different coordinate systems, we can observe whether or not each triangulation converges in the same fashion, or if it matters depending on the formulas. If we do discover that all systems behave the same way, we would then like to focus on one system of coordinates, most likely Euclidean, as that would be easier to grasp and be more computational.

7 Conclusions

We learned a lot of things along the way. At the beginning, we all had varying familiarities with college geometry; some of us had not touched the stuff since high school. But we are excited to see where Dr. Glickenstein's project is headed and we hope to remain a part of it in the months and years to come.

We would like to thank Dr. David Glickenstein for having us on this project, Dr. Robert Indik and the University of Arizona Math Department for their help and support, and the VIGRE foundation for making this possible.

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] Wolfram Mathworld. http://mathworld.wolfram.com/.

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

[9] Wikipedia. Euler characteristic. http://en.wikipedia.org/wiki/Euler_characteristic.

8 Appendix

8.1 Derivation of Eq. (6)

We used the criteria

$$f(\tilde{r_1}, \tilde{r_2}, \ldots, \tilde{r_n}) = \prod \tilde{r_i} = \prod \alpha r_i = \alpha^n \prod r_i = C$$

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 $\frac{dr_i}{dt} = -K_i r_i$ from Eq. 1 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

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

However, since the original product is a constant, we have $\frac{df}{dt} = 0$. 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. (6)

8.2 Remarks on Runge-Kutta method for solving Eq. (8)

The method used by Moreau in [8] 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 negate this and cause the system to go haywire.

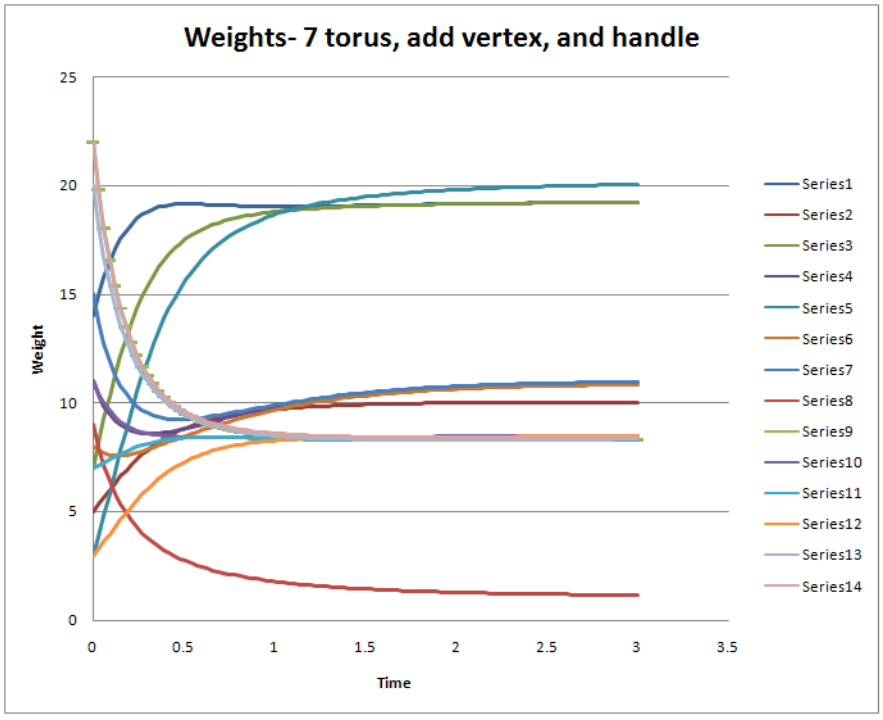

Figure 10: An example of how morphs can change the asymptotic behavior of vertices.

8.3 Code

calcFlow

```
void calcFlow(char* fileName, double dt ,double *initWeights,
int numSteps, bool adjF)
{
  int p = Triangulation::vertexTable.size(); // The number of vertices.
  double ta[p],tb[p],tc[p],td[p],z[p]; // Temporary arrays to hold data in.
  int    i,k; // ints used for "for loops".
  map<int, Vertex>::iterator vit;
  map<int, Vertex>::iterator vBegin = Triangulation::vertexTable.begin();
  map<int, Vertex>::iterator vEnd = Triangulation::vertexTable.end();
  double weights[p][numSteps];
  double curvatures[p][numSteps];

  ofstream results(fileName, ios_base::trunc);
  results.setf(ios_base::showpoint);
  double net = 0; // Net and prev hold the current and previous
  double prev;    //  net curvatures, repsectively.
   for (k=0; k<p; k++)
     z[k]=initWeights[k]; // z[k] holds the current weights.
   for (i=1; i<numSteps+1; i++)
   {
    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.
            prev = Triangulation::netCurvature();
       for (k=0, vit = vBegin; k<p && vit != vEnd; k++, vit++)
       // First "for loop" in whole step calculates
       {               // everything manually, prints to file.
           weights[k][i - 1] = z[k];
           double curv = curvature(vit->second);
           curvatures[k][i - 1] = curv;
           net += curv;
```

```
        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.
        vit->second.setWeight(z[k]+ta[k]/2);
    for (k=0, vit = vBegin; k<p && vit != vEnd; k++, vit++)
    {
         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;
}
for(k=0, vit = vBegin; k<p && vit != vEnd; k++, vit++)
{  //Print results
   results << setprecision(6);
   results << left << "Vertex: " << left << setw(4)<< vit->first;
```

```
      results << right << setw(3) << "Weight";
      results << right << setw(10) << "Curv";
      results << "\n-----------------------------\n";
      for(int j = 0; j < numSteps; j++)
      {
              results << left <<  "Step " << setw(7) << (j + 1);
              results << left << setw(12) << weights[k][j];
              results << left << setw(12) << curvatures[k][j] << "\n";
      }
      results << "\n";
   }
   results.close();
}
```

2-2 Flip

```
void flip(Edge e)
{
     //start out by naming every object that is local to the flip
     Face f1 = Triangulation::faceTable[(*(e.getLocalFaces()))[0]];
     Face f2 = Triangulation::faceTable[(*(e.getLocalFaces()))[1]];

     vector<int> sameAs;
     vector<int> diff;

     Vertex va1 = Triangulation::vertexTable[(*(e.getLocalVertices()))[0]];
     Vertex va2 = Triangulation::vertexTable[(*(e.getLocalVertices()))[1]];

     diff = listDifference(f1.getLocalVertices(), f2.getLocalVertices());
     if(diff.size() == 0)
     throw string("Invalid move, operation canceled");
     Vertex vb1 = Triangulation::vertexTable[diff[0]];
     diff = listDifference(f2.getLocalVertices(), f1.getLocalVertices());
     Vertex vb2 = Triangulation::vertexTable[diff[0]];

     sameAs = listIntersection(va1.getLocalEdges(), vb1.getLocalEdges());
     Edge ea1 = Triangulation::edgeTable[sameAs[0]];
     sameAs = listIntersection(va2.getLocalEdges(), vb1.getLocalEdges());
     Edge eb1 = Triangulation::edgeTable[sameAs[0]];
```

```
sameAs = listIntersection(va1.getLocalEdges(), vb2.getLocalEdges());
Edge ea2 = Triangulation::edgeTable[sameAs[0]];
sameAs = listIntersection(va2.getLocalEdges(), vb2.getLocalEdges());
Edge eb2 = Triangulation::edgeTable[sameAs[0]];

sameAs = listIntersection(f1.getLocalFaces(), ea1.getLocalFaces());
Face fa1 = Triangulation::faceTable[sameAs[0]];
sameAs = listIntersection(f1.getLocalFaces(), eb1.getLocalFaces());
Face fb1 = Triangulation::faceTable[sameAs[0]];
sameAs = listIntersection(f2.getLocalFaces(), ea2.getLocalFaces());
Face fa2 = Triangulation::faceTable[sameAs[0]];
sameAs = listIntersection(f2.getLocalFaces(), eb2.getLocalFaces());
Face fb2 = Triangulation::faceTable[sameAs[0]];

//removals
Triangulation::vertexTable[(va1.getIndex())].removeVertex(va2.getIndex());
Triangulation::vertexTable[(va2.getIndex())].removeVertex(va1.getIndex());
Triangulation::vertexTable[(va1.getIndex())].removeEdge(e.getIndex());
Triangulation::vertexTable[(va2.getIndex())].removeEdge(e.getIndex());
Triangulation::vertexTable[(va1.getIndex())].removeFace(f2.getIndex());
Triangulation::vertexTable[(va2.getIndex())].removeFace(f1.getIndex());
Triangulation::edgeTable[(e.getIndex())].removeVertex(va1.getIndex());
Triangulation::edgeTable[(e.getIndex())].removeVertex(va2.getIndex());
for(int i = 0; i < e.getLocalEdges()->size(); i++)
{
   Triangulation::edgeTable[(e.getIndex())]
    .removeEdge((*(e.getLocalEdges()))[i]);
}
for(int i = 0; i < va1.getLocalEdges()->size(); i ++)
{
   Triangulation::edgeTable[(*(va1.getLocalEdges()))[i]]
    .removeEdge(e.getIndex());
}
for(int i = 0; i < va2.getLocalEdges()->size(); i ++)
{
   Triangulation::edgeTable[(*(va2.getLocalEdges()))[i]]
    .removeEdge(e.getIndex());
}
```

```
Triangulation::edgeTable[(eb1.getIndex())].removeFace(f1.getIndex());
Triangulation::edgeTable[(ea2.getIndex())].removeFace(f2.getIndex());
Triangulation::faceTable[(f1.getIndex())].removeVertex(va2.getIndex());
Triangulation::faceTable[(f2.getIndex())].removeVertex(va1.getIndex());
Triangulation::faceTable[(f1.getIndex())].removeEdge(eb1.getIndex());
Triangulation::faceTable[(f2.getIndex())].removeEdge(ea2.getIndex());
Triangulation::faceTable[(f1.getIndex())].removeFace(fb1.getIndex());
Triangulation::faceTable[(fb1.getIndex())].removeFace(f1.getIndex());
Triangulation::faceTable[(f2.getIndex())].removeFace(fa2.getIndex());
Triangulation::faceTable[(fa2.getIndex())].removeFace(f2.getIndex());

//additions
Triangulation::vertexTable[(vb1.getIndex())].addVertex(vb2.getIndex());
Triangulation::vertexTable[(vb2.getIndex())].addVertex(vb1.getIndex());
Triangulation::vertexTable[(vb1.getIndex())].addEdge(e.getIndex());
Triangulation::vertexTable[(vb2.getIndex())].addEdge(e.getIndex());
Triangulation::vertexTable[(vb1.getIndex())].addFace(f2.getIndex());
Triangulation::vertexTable[(vb2.getIndex())].addFace(f1.getIndex());
Triangulation::edgeTable[(e.getIndex())].addVertex(vb1.getIndex());
Triangulation::edgeTable[(e.getIndex())].addVertex(vb2.getIndex());
for(int i = 0; i < vb1.getLocalEdges()->size(); i ++)
{
        Triangulation::edgeTable[(e.getIndex())]
         .addEdge((*(vb1.getLocalEdges()))[i]);

        Triangulation::edgeTable[(*(vb1.getLocalEdges()))[i]]
         .addEdge(e.getIndex());
}
for(int i = 0; i < vb2.getLocalEdges()->size(); i ++)
{
        Triangulation::edgeTable[(e.getIndex())]
         .addEdge((*(vb2.getLocalEdges()))[i]);

        Triangulation::edgeTable[(*(vb2.getLocalEdges()))[i]]
         .addEdge(e.getIndex());
}
Triangulation::edgeTable[(ea2.getIndex())].addFace(f1.getIndex());
Triangulation::edgeTable[(eb1.getIndex())].addFace(f2.getIndex());
```

```
        Triangulation::faceTable[(f1.getIndex())].addVertex(vb2.getIndex());
        Triangulation::faceTable[(f2.getIndex())].addVertex(vb1.getIndex());
        Triangulation::faceTable[(f1.getIndex())].addEdge(ea2.getIndex());
        Triangulation::faceTable[(f2.getIndex())].addEdge(eb1.getIndex());
        Triangulation::faceTable[(f1.getIndex())].addFace(fa2.getIndex());
        Triangulation::faceTable[(fa2.getIndex())].addFace(f1.getIndex());
        Triangulation::faceTable[(f2.getIndex())].addFace(fb1.getIndex());
        Triangulation::faceTable[(fb1.getIndex())].addFace(f2.getIndex());

}
```

About the authors

Alex Henniges is a junior double majoring in Math and Computer Science. Plus he's cool. He is the front liner of the programming aspect of this research.

Thomas Williams is a senior in Comprehensive Mathematics with a minor in Computer Science and a strong background in Math Education. He's cool, too. He made substantial contributions to the coding and development of this project.

Mitch Wilson is a senior majoring in Applied Math and Mechanical Engineering. He's ok. He made the pretty pictures in this report, mostly using MATLAB.