Generating and applying triangulations to Delaunay surfaces and combinatorial Ricci flows

Alex Henniges
Thomas Williams
Mitch Wilson

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

August 8, 2008

Table of Contents

1 Introduction **3**

2 Triangulations **3**

3 Delaunay triangulations **6**

4 Combinatorial Ricci flow **6**
4.1 Definitions and Equations 8
4.2 Programming Ricci flow 11
4.3 Initial testing and results 12
4.3.1 First tests 13
4.3.2 Specific cases 15
4.3.3 Convergence speeds 17

5 Spherical and hyperbolic Ricci flow **20**
5.1 Spherical flow 20
5.2 Hyperbolic flow 25
5.3 Comparison of Systems 26

6 Future work **28**
6.1 Linking Delaunay to Ricci flow 28
6.2 3-D 29
6.3 Circle packing expansions 29

7 Conclusion **29**

8 Appendix **32**
8.1 Derivation of Eq. (5) 32
8.2 Remarks on Runge-Kutta method for solving Eq. (7) 33
8.3 Data Plots 33
8.4 Code Examples 36

1 Introduction

The purpose of this paper is to explore and entend upon the use of triangulaitions in Delaunay surfaces and combinatorial Ricci flow. These concepts will be addressed under an analytical structure by designing a system to represent the triangulations and provide meaningful data. A large collection of data has not yet been compiled on these concepts and our goal is for such data to solidify as well produce theories in this fairly unexplored area of mathematics. The program used for this purpose will be written in C++.

This paper will begin in $\oint 2$ with an introduction to triangulations and any related definitions. Also, it will explain how our program is structured to meet its function. We will then present our research on Delaunay surfaces. That will be followed by an explanation of combinatorial Ricci flow along with the results from our initial experiments. In $\oint 5$ we continue the exploration of Ricci flow under different geometries. Lastly, we will provide areas of future work on this subject.

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 building them solely out of triangles. Since any regular polygon can be broken up into triangles, we can essentially represent any surface using this method.

For an n-dimensional space, this triangulation, written $\tau = \tau_0, \tau_1, ..., \tau_n$ consists of lists of simplices σ^k, where the super-script denotes the dimension of

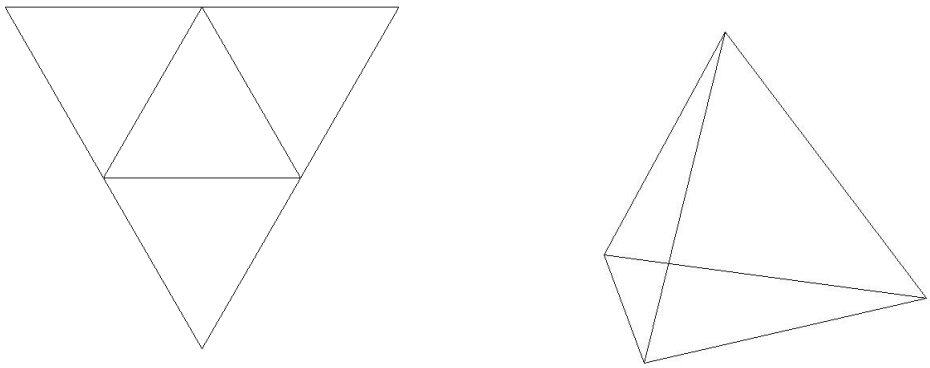

Figure 1: An example of a triangulation. The triangles can be folded up into the boundary of a tetrahedron.

the simplex and τ_k is the list of all k-dimensional simplices $\sigma^k = i_0, ..., i_k$ [4]. We shall refer to 0-dimensional simplices as vertices, 1-dimensional simplices as edges, 2-dimensional simplices as triangles or faces, and 3-dimensional simplices as tetrahedra. For this project, we will only need to discuss triangulations of 2-dimensional spaces, or surfaces.

In order to describe our surfaces we must make a definition for each individual simplex. For our data structure, we decided on creating a list of references for each simplex to give it definition 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" slightly different for each simplex. To begin with, we say that an edge is defined by two vertices and a face is defined by three vertices and three edges. Then, we can say that a vertex is local to any edge that it is in the definition of and any face that it is in the definition of, as well as any vertex that it shares an edge in common with. An edge is local to any vertex that it is defined by and any face that it is in the definition of, as well as any edge that it shares a vertex in common with. A face is local to any vertex it is defined by, any edge that it is defined by, and any face that it shares an edge in common with. These are the definitions for locality of simplices that we decided upon, creating three different lists for each simplex.

As well as lists of references to other simplices, each simplex also has other information important to the formation of 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 w_i. We think of the weight of a given vertex to be the square of the radius of a circle centered at that vertex. We are then able to state that 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. We use this particular 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. Additionally, each edge also has a center defined by the weights and the length of the edge.

Based on these centers, we are able to define the *local lengths* of an edge, as well as the *dual* of a given simplex, a concept that will become more important later on. 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 $\mathrm{C}(\{i, j\})$, the center of $\{i, j\}$. It also holds that

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

for any face $\{i, j, k\}$. Based on this condition

First we will define a dual for a given edge. Let us refer to any edge, along with the faces of which it forms an intersection, as a *hinge*. We are able to embed any hinge in the plane.

$$h = \frac{d_{13} - d_{12}\cos(\theta)}{\sin(\theta)}$$

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. 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 makes it quick and easy to move from one simplex to another. As seen in Figure **??**, all simplices are assumed to have lists of references to other simplices, what we call local simplices, broken down by dimension. For the two-dimensional case, 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 is made up 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.

3 Delaunay triangulations

- Definition of Delaunay triangulation
- Plane generation algorithm with necessary equations
- Math, flips, duals, negative triangles, equations, oh my!
- Verifications, duals

One special type of triangulation is known as a Delaunay Triangulation.

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. 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

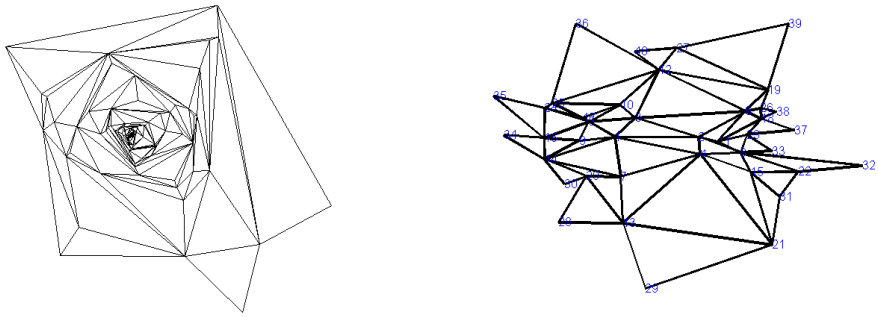

Figure 2: Two examples of triangulations using our *generateTriangulation* program, an early version (left) and a modern version (right).

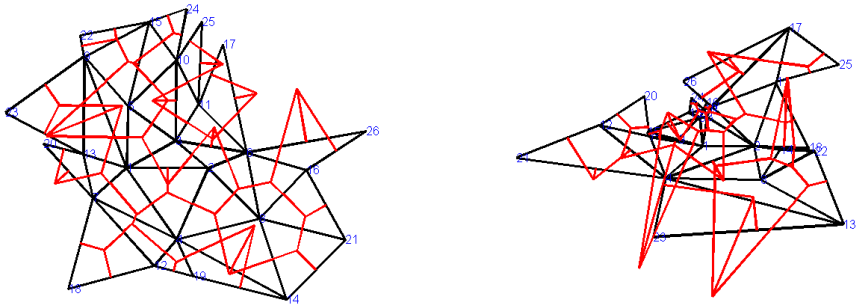

Figure 3: A non-weighted (left) and weighted (right) triangulation with dual lengths added.

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.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 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]. 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 the Ricci flow Hamilton had defined. That the subject is still very new makes this research project exciting.

4.1 Definitions and Equations

We define a manifold to be a topological space where every neighborhood is locally Euclidean. This means that around any point in the space it appears similar to the sphere locally. What this means for our triangulated surfaces is that there are no borders. We also refer to this as a closed triangulation.

When talking about closed, triangulated surfaces an important characteristic is known the Euler characteristic χ, defined by the equation $\chi = V - E + F$, 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 toplogical property that is more loosely known as the number of "holes" in the surface. 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 circle packed, vertices can be too large or too small, and the resulting geometry can be somewhat intractable. 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 combinatorial Ricci flow to the sytem. On a discrete surface, this equation allows the weights to

change over time [2]. We present the equation to the reader, which 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 the vertex i. We use the words "radius" and "weight" interchangeably, denoting the length of the radius that surrounds a vertex i in circle-packing. 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 we are performing multiple non-linear differential equations as variables depend on the weights which change over time, we can probably not solve them explicitly.

A potential issue we noted is that, based on the equation, is it possible that the radii could continually decrease. 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 by the same amount, but the curvature of each vertex would still remain π because the curvature is not affected by uniform scaling. The radii would continue to decrease until they approach zero length. We have to address that issue since computers don't like working with numbers near zero, as in the denominator of the arccosine function. Numerical instability may occur. To avoid this issue, let us resize the length of each radius by a scalar, α. We denote each scaled length by $\tilde{r_i}$ and equate as

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

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

$$\begin{aligned} \frac{d\tilde{r_i}}{dt} &= \frac{d(\alpha r_i)}{dt} = \alpha \frac{dr_i}{dt} + r_i \frac{d\alpha}{dt} \\ &= -\alpha K_i r_i + \frac{\tilde{r_i}}{\alpha} \frac{d\alpha}{dt} \\ &= -\tilde{K_i} \tilde{r_i} + \frac{\tilde{r_i}}{\alpha} \frac{d\alpha}{dt}. \end{aligned} \tag{3}$$

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 criterium:

$$f(\tilde{r_1}, \tilde{r_2}, \ldots, \tilde{r_n}) = \prod \tilde{r_i} = \prod \alpha r_i = C, \text{ a constant.} \tag{4}$$

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. (4) with respect to time we find that

$$\frac{d(\log \alpha)}{dt} = \frac{\text{sum of all curvatures}}{\text{number of vertices}} = \overline{K}, \text{average curvature.} \tag{5}$$

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 and the Euler characteristic. We know that the sum of angles from each vertex is 360°, or 2π. However, we also 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 trends in basic triangulations. 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 **??** we note this true for all polyhedra listed. Let us use this substitution and 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 $\frac{\sum K_i}{|V|} = \frac{2\pi\chi}{|V|}$ where $|V|$ is the number of vertices. This is also noted in [2]. Plugging this information back into Eq. (3) we determine that

$$\frac{d\tilde{r_i}}{dt} = -\tilde{K_i}\tilde{r_i} + \overline{K}\tilde{r_i} = (\overline{K} - \tilde{K_i})\tilde{r_i} \tag{6}$$

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:

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

This is known as normalized Ricci flow, as discussed by Chow and Luo in their paper [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 $\frac{dr_i}{dt} = 0$ for $i = \{1, 2, 3, 4\}$.

4.2 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, (7), should be used. Otherwise, the standard equation (1) 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 1.

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

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 1: Two steps of a Ricci flow

the speed of the non-adjusted flow in comparison, it became clear that the calculation of total 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 ∮ 8.4.

4.3 Initial testing and results

We have some expectations for combinatorial Ricci flow over two-dimensional Euclidean surfaces. Cases like the boundary of 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 (7) the weights are expected to converge to positive constants.

There are other expected behaviors. 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, what we call the *product area*. Also, it is predicted that the total curvature of a 2-manifold surface should remain a constant determined by its genus.

For the 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 develop methods to test and find any errors in our code.

Initial checks for accuracy:

1. Boundary of tetrahedron will converge to equal weights.
2. Constant *product area*.
3. Constant total curvature.

4.3.1 First tests

The program for the Ricci flow was tested by beginning with the simplest cases, and then explored with as many different possibilities as we could conceive of to try to find anomalies. The first test was the boundary of 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 *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 a 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 extremes amongst the initial weights causes a different end result.

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 2: Adding three vertices to a Tetrahedron

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 2, 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 never less than four, while vertices 4 and 6 each have a local vertex with degree 3. See Figure 12 in the Appendix for an illustration of weights over time.

4.3.2 Specific cases

Example: Adding a vertex to a one-holed torus. See figure 4.

By inserting a vertex within a triangulation for a torus, we are essentially creating a bump on the torus and then observe what happens as we run it through 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. (4). 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.

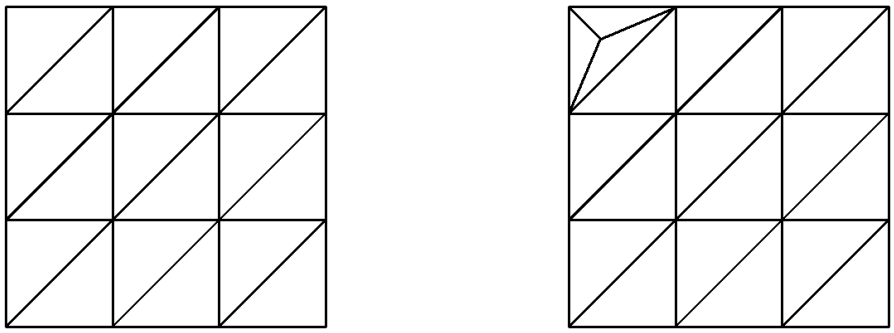

Figure 4: A triangulation of the torus, and the addition of a new vertex. This is a 1-3 flip.

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 $\sim \pi$ 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.

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 observed that the vertices of a triangulation all converge to $\dfrac{2\pi\chi}{|V|}$ curvature. 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

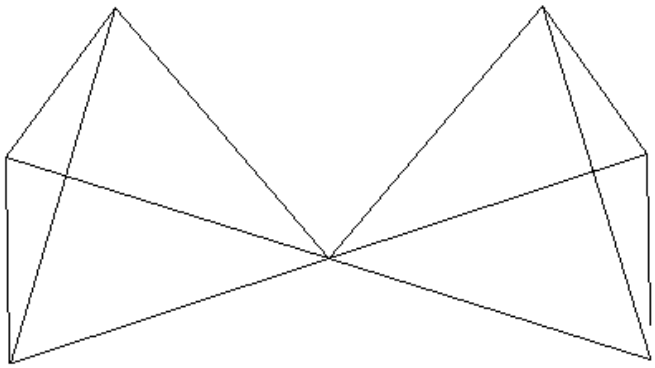

Figure 5: Two tetrahedra conjoined at a vertex.

not provide large enough triangulations for fourth degree vertices. A data plot of this similiar situation with a genus 6 triangulation is provided in § 8.3.

Example: Two tetrahedra connected at a vertex. See Figure 5

It turns out $\chi = 7$ 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 total curvature equaled $6\pi = 2\pi\chi$. Technically, this example contradicts our notion of a manifold, but we felt it useful in validating our program procedure.

4.3.3 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. It was unclear what set weights could have an equal 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 time consumming 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 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 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, 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.

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 6: Summary of convergence data for varying criteria

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 the reader as they may be unfamilar 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 edge lengths. 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 producable on a sphere of radius 1, the unit sphere.
- By default, we imply the geodesic of a spherical triangle, using the shortest distance on the surface between any two vertices. As a result, the sum of the weights of any triangle cannot exceed π. In addition to contradicting the previous definition, such a situation can lead to undefined calculations of our angles, the equation of which is provided below.

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 this is analogous to the Euclidean version.

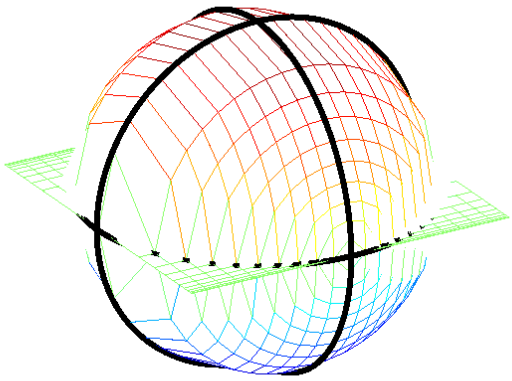

Figure 7: An illustration of a spherical octahedron through the X-Y plane. Each octet of the sphere surface, outlined in black, is the face of one triangle.

With $\sin(x) \approx x$ and $\cos(x) \approx 1 - \frac{x^2}{2}$ for x small, we get

$$\begin{aligned}
\cos(\angle C) &= \frac{\cos(c) - \cos(a)\cos(b)}{\sin(a)\sin(b)} = \frac{(1 - \frac{c^2}{2}) - (1 - \frac{a^2}{2})(1 - \frac{b^2}{2})}{ab} \\
&= \frac{\frac{a^2}{2} + \frac{b^2}{2} - \frac{c^2}{2} + \frac{a^2 b^2}{4}}{ab} \\
&= \frac{a^2 + b^2 - c^2}{2ab} + \frac{ab}{4} \approx \frac{a^2 + b^2 - c^2}{2ab} \text{ as } \frac{ab}{4} \approx 0 \text{ for } a, b \text{ small}
\end{aligned}$$

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

$$\frac{dr_i}{dt} = -K_i \sin(r_i). \tag{8}$$

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:

- $\frac{dr_i}{dt} = (\overline{K} - K_i)\sin(r_i)$
  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 tetrahdron, 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 in Chow and Luo's paper [2]. Since the surface area went to 4π, then the average curvature went to zero. While we liked the end result of our triangulation trials, the system failed far too often to be reliable.

- $\frac{dr_i}{dt} = (\hat{K} - K_i)r_i$
  Following the criteria we used to obtain our normalized Euclidean flow, we began experimenting and used the cases $\tilde{r_i} = \alpha r_i$ 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. 8.

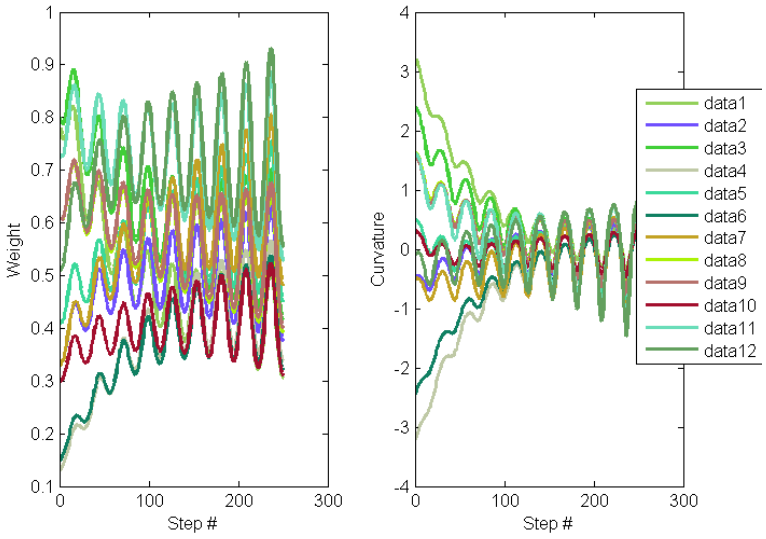

Figure 8: An example using our notion of $\hat{K}$. Weights and curvatures were unable to converge, and crashed the program shortly after.

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

$$\frac{dr_i}{dt} = \overline{K} r_i - K_i \sin(r_i). \tag{9}$$

In examining its behavior with various triangulations, we found that this one works very well. We're not completely sure why, but it does. We would like to be able to mathematically determine the effectiveness of this equation. We find that spheres converge to zero curvature and weights first group together and then approach optimal weights. See Figure 9. 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 dependant 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 $r_i = \frac{1}{2}\arccos(-\frac{1}{3}) \approx 0.9553$. The only downsides we found with this equation were that it was still possible for the system 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. 9, we see that with the dt size given of 0.5, it would take over 500 steps for the system to reach equilibrium, and this would be much longer with the dt used in $\oint$ **??**.

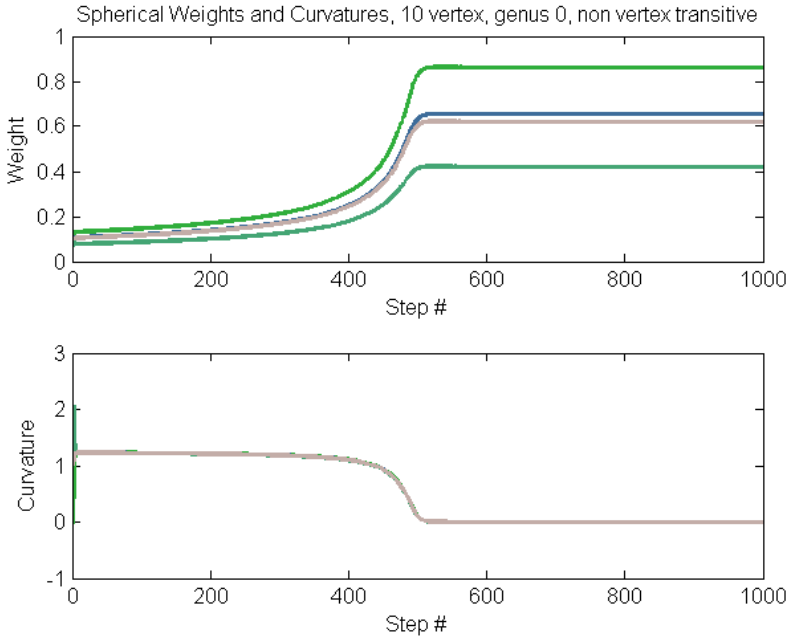

Figure 9: An example of a solution using Eq. (9). Starting off with equal initial weights of 0.1, vertices merge into one of four groups as they approach uniform curvature relatively quickly. As curvature drops to zero, slowly at first but more forcefully as time passes, the weight groups separate from 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. 10. This representation is often called the Poincaré disk, named after the same Poincaré as in §4.

There are several interesting properties with the hyperbolic plane. We now have that for a line l and a point P not on that line, there are an infinite number of lines throught 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 found by following a curved path along a circle centered at infinity going through both points. A more thorough explanation of 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

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

Most importantly, the equation for combinatorial Ricci flow is now

$$\frac{dr_i}{dt} = -K_i \sinh(r_i) \tag{10}$$

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.

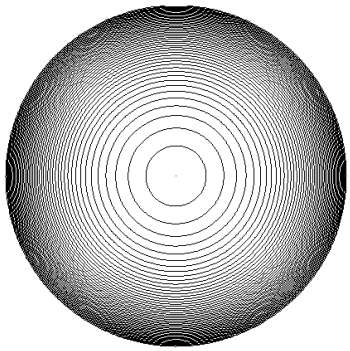

Figure 10: An illustration of the hyperbolic plane using the Poincaré disk. All concentric circles are evenly spaced apart in a Euclidean background, but the outer edge of the disk approaches infinity, so the circles appear closer together. A similar way to think of it is to imagine looking at the contour graph of $z = x^2 + y^2$ from the point (0,0,-1).

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 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 non-normalized equations, (1), (8), and (10). 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 group together somewhat as in the normalized spherical case, and then continue 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 sphericalCalcFlow 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 conlude, we generalize 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. Spherical combinatorial Ricci flow is said to 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

Our generated triangulations can be akinned to our original notion of circle packing. We would like to close our triangulations and be able to properly run them through our Ricci flows. The major issue right now is that no vertices on the boundary connect to each other. We discovered that, for

our triangulations, we always obtained a χ value of 1, so we would have to perform some additional manipulations on the end result to obtain a proper manifold.

6.2 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 David Glickenstein 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.

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 be able to move from one triangle to another seamlessly. We can lay groundwork for that, and when this project is able to jump to 3-D modeling, we hope that this will help. We would also like to adapt our code into OpenGL, software that enables us to build 3-D surfaces using high-tech simulators here at the University of Arizona.

6.3 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 their angle of intersection. See Figure 11. 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].

7 Conclusion

We had a lot of goals for this project. For one, we are the first group to work with Dr. David Glickenstein and he has his own long-term goals. As the

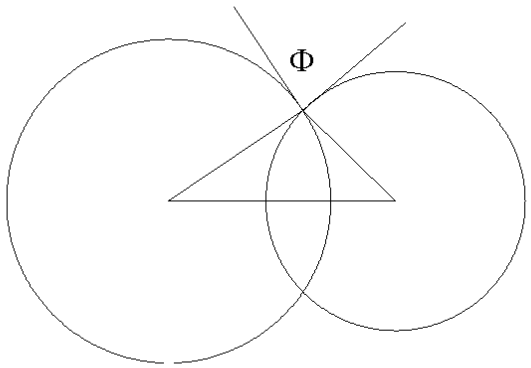

Figure 11: An example of relaxing circle packing and introducing Φ.

initial builders for these goals, we feel that we have laid a good foundation that can be easily built upon by future undergraduate students. 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. Another goal was to properly implement Ricci flow for 2-dimensional manifolds. While there are a number of extensions that still need to be implemented, we are pleased that the flow is functional and providing useful data. Our jump to weighted Delaunay triangulations enabled us to make our code more adaptable, as well as be able to generate our own systems with desired properties. We also feel we have learned an enormous amount over the course of only a few weeks. At the beginning, we all had varying familiarities with college geometry; some of us had not had a course in geometry since high school. Lastly, the experience from writing a research report will undoubtedly help us in our future endeavors. 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 National Science Foundation VIGRE #_____.

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.

8 Appendix

8.1 Derivation of Eq. (5)

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

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

If we assume the 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. (5)

8.2 Remarks on Runge-Kutta method for solving Eq. (7)

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.

8.3 Data Plots

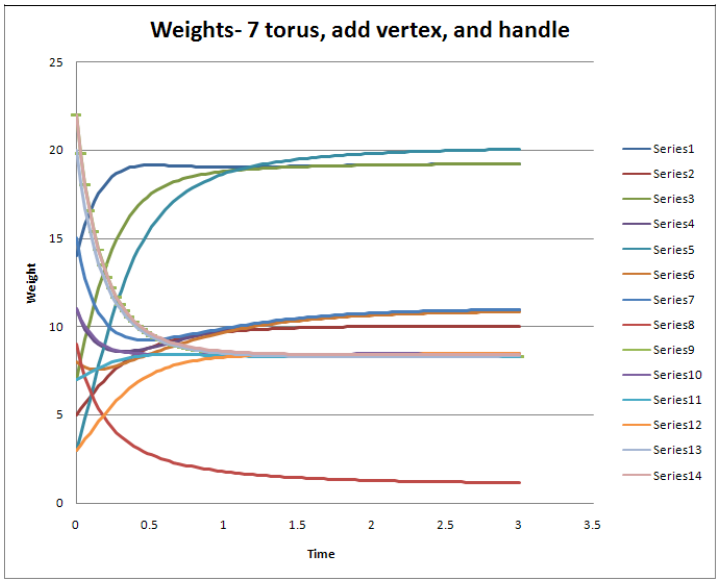

Figure 12: An example of how morphs can change the asymptotic behavior of vertices. In this case we saw the weights of some vertices change concavity.

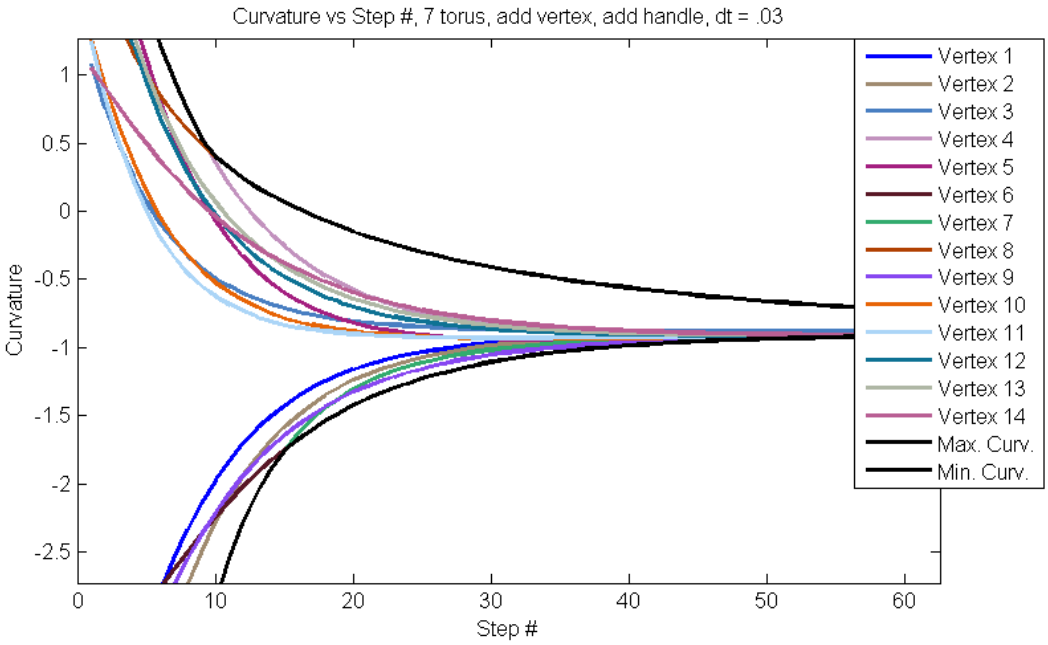

Figure 13: An example of curvatures over time. While they do converge to the same curvature, the vertex with the maximum or minimum curvature may change. This is a separate trial than that producing Fig. 12

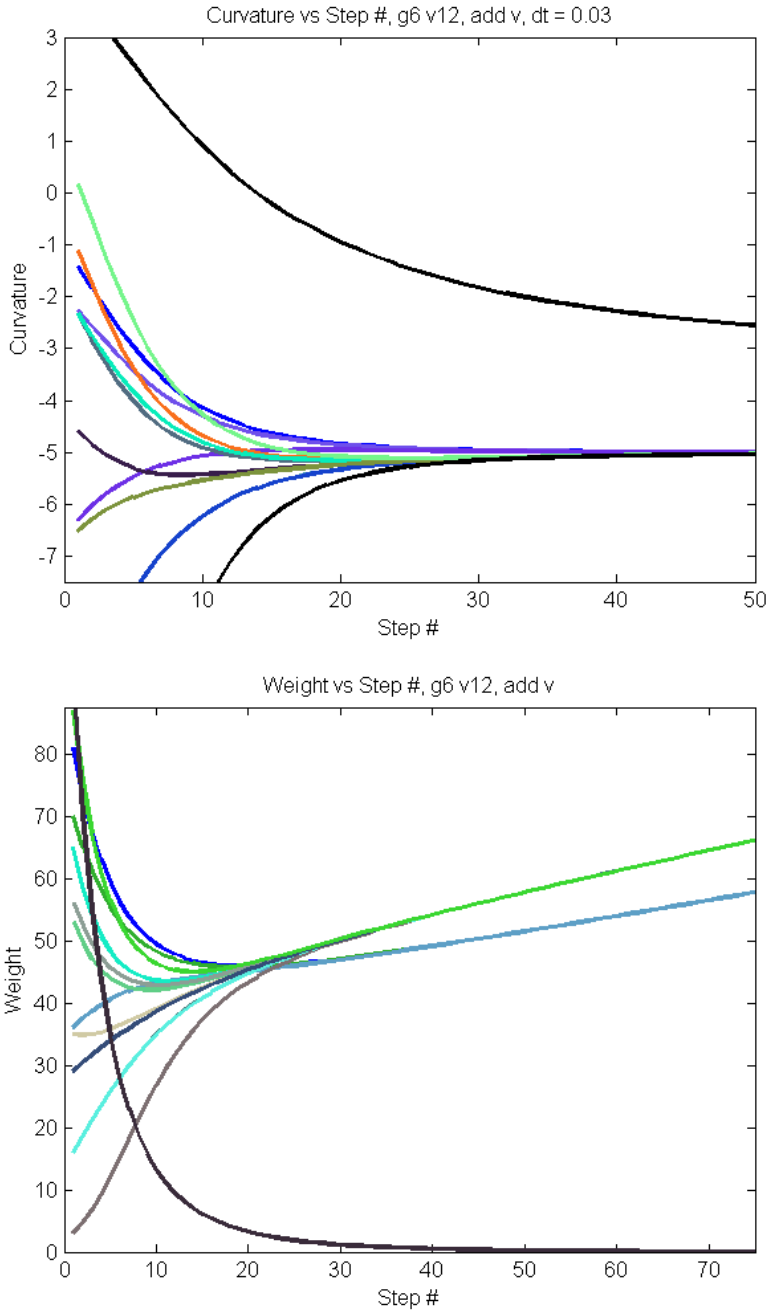

Figure 14: An example of adding a vertex to a genus 6 surface. One of the curvatures in unable to drop below $-\pi$, and as a result, its weight is pushed to almost zero. Other vertices group together to compensate for this behavior.

8.4 Code Examples

- **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. 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-
- Thomas Williams-
- Mitch Wilson- mjw@email.arizona.edu

Project website: http://code.google.com/p/geocam