University of Nebraska at Omaha
DigitalCommons@UNO

Student Work

5-2017

Applications of Graph Embedding in Mesh Untangling

Jake Quinn
University of Nebraska at Omaha

Follow this and additional works at: https://digitalcommons.unomaha.edu/studentwork

Part of the Computer Sciences Commons

Recommended Citation
Quinn, Jake, "Applications of Graph Embedding in Mesh Untangling" (2017). *Student Work*. 2916.
https://digitalcommons.unomaha.edu/studentwork/2916

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.

Applications of Graph Embedding in Mesh Untangling

A Thesis
Presented to the
Department of Computer Science

and the

Faculty of the Graduate College
University of Nebraska
In Partial Fulfillment
of the Requirements for the Degree
Master of Science, Computer Science

University of Nebraska at Omaha
by
Jake Quinn
May 2017

Supervisory Committee:
Dr. Sanjukta Bhowmick
Dr. Robin Gandhi
Dr. Yulia Lierler

ProQuest Number: 10271481

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10271481

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Applications of Graph Embedding in Mesh Untangling

Jake Quinn, BSCS
University of Nebraska, 2017
Advisor:
Dr. Sanjukta Bhowmick

Abstract:

The subject of this thesis is mesh untangling through graph embedding, a method of laying out graphs on a planar surface, using an algorithm based on the work of Fruchterman and Reingold[1]. Meshes are a variety of graph used to represent surfaces with a wide number of applications, particularly in simulation and modelling. In the process of simulation, simulated forces can tangle the mesh through deformation and stress. The goal of this thesis was to create a tool to untangle structured meshes of complicated shapes and surfaces, including meshes with holes or concave sides. The goals of graph embedding, such as minimizing edge crossings align very well with the objectives of mesh untangling. I have designed and tested a tool which I named MUT (Mesh Untangling Tool) on meshes of various types including triangular, polygonal, and hybrid meshes.

Previous methods of mesh untangling have largely been numeric or optimization-based. Additionally, most untangling methods produce low quality graphs which must be smoothed separately to produce good meshes. Currently graph embedding techniques have only been used for smoothing of untangled meshes. I have developed a tool based on the Fruchterman-Reingold algorithm for force-directed layout[1] that effectively untangles and smooths meshes simultaneously using graph embedding techniques. It can untangle complicated meshes with irregular polygonal frames, internal holes, and other complications that previous methods struggle with. The MUT does this by using several different approaches: untangling the mesh in stages from the frame in and anchoring the mesh at corner points to stabilize the untangling.

Table of Contents:

List of Multimedia Objects:

1. Introduction

Graph theory is an influential field of mathematics that can revolutionize production and industry as well as influence modern engineering. Take for example the study of medicine. It is difficult and dangerous to observe the effects of stress on a human heart, but it is possible to simulate this scenario using meshes. Car manufacturers have long relied on crash tests to determine the safety of their vehicles when subjected to extreme stress, which requires the destruction of potentially many expensive vehicles. With the rise of more powerful computing, simulation has provided an alternative to real-world testing in many industries such as medicine, engineering, and others.

It is impossible to perfectly replicate a physical object digitally. To do this, the location of every atom would have to be determined and recorded, and to simulate any interaction with the object, algorithms would have to be executed on every one of those points. Instead, when modelling an object, a number of points are selected from the surface of the object. Adjacent points are then connected, and the resulting structure is called the mesh of the object. A mesh is a specific type of graph, which is itself a mathematical structure most simply defined as being a collection of vertices and edges together with a relationship stating that every edge of the graph connects either two vertices or a vertex to itself. More formally a mesh is a graph that defines the shape of an object in modelling and includes polygonal faces formed by components of the graph. Additionally, a mesh is a type of simple graph, in which there are no edges connecting a vertex to itself.

A mesh in a physical simulation is subject to any number of forces upon itself. These forces are represented as a system of equations. The force on any given point is defined by a partial differential equation. In the continuous domain, these

equations are not solvable, so in order to perform calculations on a mesh, the partial differential equations must be transformed into the discrete domain where they become solvable. This system of equations can be treated as a matrix, and can be solved using matrix operations.

This approach is not without challenges. For the matrix to be easily solved, the matrix must be well-conditioned. For the matrix of a mesh to be well-conditioned, the mesh cannot be tangled, and the shapes of the components, or bounded regions, should be as regular as possible. When affected by forces or other influences, a mesh can become tangled in order to conform with the surface it represents. When the mesh becomes tangled it could simply be re-formed based on the new locations of the vertices; however this loses all information about the original connections. In many circumstances, this is undesirable. Imagine a muscle that has torn; simply sewing it back together wherever it ends up would likely permanently damage it.

To summarize, in order to perform the operations necessary for simulation on a mesh, it must be well-conditioned, with the most significant factor of this being that the mesh must be untangled. The challenge then is to untangle the mesh. In particular, this is difficult when the boundary of the mesh is not a convex shape, or is otherwise irregular. There are a number of paradigms and methodologies that have been explored in the pursuit of effective and efficient mesh untangling. Traditional approaches include numeric approaches, which seek to use linear algebra techniques to make the mesh well-conditioned. However, this tends to be very limited depending on the type of mesh components involved. Triangular meshes must be treated very differently from quadrilateral meshes, and so on.

This thesis will present the Mesh Untangling Tool, or MUT, which provides a different paradigm. My research opted for a holistic approach, treating a mesh as one

would any other graph and using graph embedding to untangle the mesh. Graph embedding is a method of laying out a graph on a surface. This strategy is viable because the goals of mesh untangling are very similar to the goals of graph embedding. Notably, both seek to avoid edge crossings. In addition, graph embedding seeks to normalize angle measures and side lengths of components within the graph, which leads to a more untangled mesh.

In the remainder of this thesis, I will present the MUT, the implementation, and theory behind it, as well as experimental results in the space of two-dimensional mesh untangling for a myriad variety of mesh types. The MUT provides significant advances over previous work using graph embedding as it addresses weaknesses of the graph embedding method such as handling irregular mesh boundaries as well as cases when internal vertices may be tangled outside of the main frame of the mesh. Additionally, we will address the limitations of the MUT and discuss future research to be carried out using this paradigm.

2. Background:

To understand the MUT and its function, a basic understanding of graph theory and other concepts is necessary. The necessary background for understanding the theory and basic implementation of the MUT is presented below.

2.1: Terminology

-Graph: Mathematically defined as "an ordered triple $G = (V, E, \varphi), where\ V \neq \emptyset, V \cap E = \emptyset, and\ \varphi: E \rightarrow P(V) is\ a\ map\ such\ that\ |\varphi(e)| \in \{1,2\} for\ each\ e \in E$"[2]. Put more simply, it is a collection of edges and vertices and the relationship between them.

-Mesh: A type of graph used to represent the surface of an object in a simulation.

-Vertex: Part of a graph represented visually by points.

-Edge: A component of a graph representing a connection between two vertices.

-Adjacency matrix: An n by n matrix where n is the number of vertices in a graph. Each element e(i,j) = 1 if there is an edge between vertices i and j.

- Partial Differential Equation: an equation containing unknown multivariable functions and their partial derivatives.

- Fruchterman-Reingold algorithm: An algorithm for better graph drawing through force-directed placement.

- Graph Embedding: A method of laying out a graph onto a surface such that there are no edge intersections.

2.2: Graph Theory

Graph theory is a field of discrete mathematics that has many applications to solve real-world issues. Graphs can be used to model many problems that exist in the physical world in order to solve them[2]. One of the fundamental concepts of graph theory is that when a real-world problem has been mapped to the domain of graph theory, solving the graph problem also provides the solution to the real-world problem.

Graph theory serves as the foundation of the MUT. In pure graph theory, a graph can be described entirely with an adjacency matrix. An adjacency matrix is a matrix with values describing which pairs of vertices share an edge. However, in the case of meshes like those dealt with by the MUT, the location of the vertices in a plane is also significant.

2.3: Fruchterman-Reingold Algorithm

The Fruchterman-Reingold algorithm, introduced in [1], formed the basis for my methods. The premise of the paper is to use a force-directed method of vertex placement in order to produce graphs that are aesthetically pleasing. The algorithm the authors developed seeks to: “Distribute the vertices evenly within the frame, minimize edge crossings, and make edge lengths uniform”[1] among other goals. The challenge here is that the processes required to make a graph aesthetically pleasing are generally NP-hard problems [3].

The basis for the Fruchterman-Reingold research has its roots in physics. According to the authors, they based their algorithm largely on work of Eades[1]. Conceptually, the graph is regarded as a physical system of rings and springs wherein the springs, representing the edges of the graph, exert force on the rings,

representing the vertices. In Eades' work, the forces seek out a state of minimal energy. The Fruchterman-Reingold algorithm takes advantage of this based on the assumption that a low energy system most closely reflects the criteria of aesthetics. In both Eades and Fruchterman-Reingold, the forces exerted on the graph are not modelled on actual physical forces such as Hooke's Law, but rather developed specifically for the desired results.

Similar to [4], the Fruchterman-Reingold algorithm restricts edges within the graph to straight lines. This not only better fits the spring metaphor but it also simplifies the concept of edge crossing by making it possible to determine if two edges cross by solving a system of linear equations. However, [4] prioritizes aesthetics over planarity, meaning that edge crossings are more permissible in [4] than in [1].

The Fruchterman-Reingold method relies on two simple requirements for vertex placement: that two vertices connected by an edge should be close together, but that no two vertices should be too close together. These are both somewhat abstract. The definitions of what is "too close" and what is "close enough" depend on the graph in question and its density. To obey these two rules, Fruchterman and Reingold looked to particle physics. Nucleons attract each other very strongly at close range, with the attraction reducing very rapidly the further apart they are. However when two nucleons are very close to one another, the strong nuclear force instead repulses the particles from one another, preventing the nuclei from collapsing. Following this principle, the Fruchterman-Reingold algorithm causes vertices that are connected by an edge to attract one another until balanced out by repulsive forces. While only vertices sharing an edge can attract, all vertices will repel one another when in close proximity.

The forces of attraction and repulsion in the Fruchterman-Reingold algorithm are based on the size of the bounding box. They define the optimal distance between vertices to be $k = C\sqrt{\frac{area}{number\ of\ vertices}}$, where C is a constant. Based on this, the authors determined that the following equations most accurately model the attractive and repulsive forces: $f_a(d) = \frac{d^2}{k}$, and $f_r(d) = -\frac{k^2}{d}$ [1] where k is the optimal distance between vertices and d is the current distance between vertices. According to the paper, these particular equations help to overcome the challenge of moving a vertex past another to overcome bad placement, which would be difficult with linear equations.

An important factor of the Fruchterman-Reingold algorithm is the bounding box, or 'frame' as it is referred to in the paper. We do not use this term in this paper, as 'frame' has a very different meaning in the MUT with respect to the Fixed-Frame type operation mode. In the Fruchterman-Reingold algorithm the dimensions of the bounding box are input by the user, and the movement of vertices is restricted within it. This is one factor that prevents the mesh from expanding too greatly. Another factor is the so-called temperature of the mesh, which decreases over time. Similar to temperature in physics, the temperature of the mesh in the Fruchterman-Reingold algorithm determines the maximum movement possible for a vertex during an iteration. Over time, the temperature decreases, reducing the overall movement in each step in an attempt to reach equilibrium.

2.4: Graph drawing

One of the simplest but most frustrating challenges in any graph drawing approach is that the pursuit of some goals of aesthetics can harm the pursuit of others. In [5], the example is given of the goals of uniform edge lengths and the avoidance of edge crossings competing. In [1] as well as the MUT, strong prioritization has been given to the minimization of edge crossings over other criteria, as even when a graph has vastly different edge lengths, the mesh can still be untangled based on the intersections of the edges. Other methods focus on the minimization rather than the elimination of all edge crossings. For example, [6] discusses the mechanics of drawing complete graphs such that certain subgraphs have no edge crossings, or edge crossings for the graph as a whole are minimized. [7] discusses a more generalized approach pertaining to minimizing edge crossings in general graphs.

In [9] another method of graph drawing is proposed wherein the vertices are all placed along a straight line and the edges are drawn as curved arcs between them. This dispenses of the effort to equalize edge lengths in favor of fewer edge crossing as well. As discussed before the curved edge model is not ideal for the purposes of mesh untangling, at least with regard to a spring model untangling paradigm. In addition, Fáry's theorem [23] states that for any planar graph, there exists an embedding in which all edges are drawn as straight line segments which do not intersect. This is proved in [23]. Due to this it is possible to avoid the significantly more difficult matter of handling curved line segments in the pursuit of untangling.

Additionally, related to the theorem in [23], [24] proves that minimizing the edge lengths of a convex planar drawing produces a unique convex polygon as well. This is similar in certain ways to logic of [1] in that it attempts to normalize

edge lengths, in this case through minimization. The Fruchterman-Reingold algorithm tends to benefit more from expansion to perform this. However, according to [25] it is an NP-hard problem to construct a planar graph with only straight line segments with a predetermined edge length[3]. According to [26] any planar graph "can be drawn on a grid of quadratic size" [26] in linear time per [27] and [28]. However, [26] deals largely with 1-planar graphs, which differ from planar graphs in that each edge can cross at most one other edge[30]. While this is intended to deal with a larger variety of graphs, with regards to the MUT the focus is on planar graphs that can be completely untangled.

The influence of [10] on the development of the Fruchterman-Reingold algorithm is clear, as it introduces the "spring embedder" concept discussed again in [3]. In the MUT the spring model system is superior to earlier models such as [11] which rely on symmetry, though both [1] and [10] also seek symmetry as a byproduct of other processes. The reason in particular why symmetry is not ideal in the MUT is that while a tangled mesh can certainly be symmetric, an untangled mesh does not have to be. While symmetry is not unwelcome, it is not sufficient for untangling.

Like the MUT, the approach discussed in [12] focuses on the minimization of edge crossings over other criteria for readability. However, in this case like many of the others, the focus is only on minimization rather than elimination. According to [13], at the time of its 2011 writing the best complexity for solving the problem of edge crossing minimization is $O(n \cdot poly(d) \cdot \log^{3/2} n)$ where d is the maximum degree of a node, poly(d) is a polynomial equation of d, and n is the number of nodes in the graph. Technically speaking, if the graph can be drawn in such a way that there are no edge crossings, such algorithms will produce a graph

without them, effectively untangling the mesh. However, not only are the methods such as those discussed in [13] and others complicated mathematically, the problem of edge crossing minimization is still considered an NP-hard problem as per [3]. A review of the process of crossing number reduction in these works and [16] as well as methods of graph planarization as in [17], [18], [19], and [20] all suggest that a heuristic-based approach would yield the maximum efficiency.

Planarization is an interesting subject with regards to mesh untangling. Determination of a planar embedding is the subject of many papers as mentioned previously. While it is not a simple subject, there is research to suggest that planar graphs can be embedded in linear time, as in [21] which builds off the work done in [22] using PQ trees, a variety of permutation based tree. Many methods for aesthetic graph drawing are based on the idea of drawing a planar graph in a particular way.

These challenges could result in lengthy computation at best, and total insolvability at worst. This is why many approaches such as those in [1] as well as [14] and [15] rely on heuristics. In [14], which builds on the work of [15], the approach taken involves the subject of simulated annealing. [14] treats graph drawing as an optimization problem and incorporates a concept from physics called annealing, in which changes to the system formed by the graph that result in a lowering of the graph's overall energy are more likely than ones that raise it when changes are random. In some ways this is similar to [1] and to the MUT as it treats the graph as a physical system with energy. In both cases the overall energy of the graph is higher when the system is more tangled, and as the graph approaches an untangled state the energy reaches lower values. However in [1]

and the MUT the changes made to the system are specifically those which reduce the overall energy and there is no element of randomness.

Ultimately the Fruchterman-Reingold method in [1] was chosen as the basis for the MUT for several reasons. The most obvious of these is the relative simplicity of the algorithm, to say nothing of how closely the definition of aesthetics presented in [1] aligns with the goals for mesh untangling. Additionally, as the Fruchterman-Reingold algorithm is very general, it was easy to expand upon and modify to meet the needs of a variety of different types of meshes.

2.5: Related Work in Mesh Untangling

The Fruchterman-Reingold algorithm is not the only method that can be used to untangle meshes. According to [31], many methods for mesh untangling are optimization-based, wherein the base untangling is performed with one optimization problem, and then the result is smoothed in another optimization problem. There are a number of methods that involve repositioning of vertices based on the sets they should belong in.

There are many different methods of mesh untangling, with varied methodology and origin. In [32] the problem is also approached as an optimization problem, in which the area of components of the graph is maximized. According to the paper this can be solved in linear time, however while effective for untangling this method tends to produce low quality meshes. The work in [33] tries a different approach with the goal of improving the quality of the mesh and untangling it concurrently. This is done by applying the smoothing optimization to the tangled mesh in such a way that it will also untangle. Overall the paper

states that this reduces the number or iterations required to reach a certain quality of mesh. The methods in [33] operate on tetrahedral meshes, which are essentially three-dimensional triangular meshes, so it is possible that the same methods would work on triangular meshes. More in line with the methods of [1] is the work in [34], which utilizes a force-directed method for the smoothing of meshes. The main difference between [34] and the MUT is that in [34] this approach is solely used for smoothing and not untangling.

Other physics-based methods such as the Fruchterman-Reingold algorithm of [1] have been explored in other research. One example of this is the spring embedder mentioned in [10]. However, as is the case [33], many such methods are intended for mesh smoothing rather than untangling as a whole. In [31] the concept of using the Fruchterman-Reingold algorithm as an untangling tool is introduced, a subject which is expanded upon in this thesis.

3. Methods and Implementation: The MUT

The MUT is the tool I have developed to untangle meshes through graph embedding. It has three modes of operation designed to handle different untangling needs effectively. It draws upon the Fruchterman-Reingold algorithm but also extends and modifies it in order to overcome some of the limitations of the original method.

3.1: Fruchterman-Reingold Influence:

The Fruchterman-Reingold algorithm is important because one can map the goals of the algorithm to the conditions of untangling a mesh. In particular, the goals of equalizing edge length, minimizing edge crossings, and distributing vertices evenly within the frame are beneficial to the task of mesh untangling. The MUT relies upon the Fruchterman-Reingold algorithm or modified variants of it to perform the untangling. However, there are some notable differences.

While in the original Fruchterman-Reingold algorithm, the concept of temperature is used to limit the spread of the vertices of the mesh, in the MUT, this is not employed. This is because it was found through experimentation that the algorithm performs satisfactorily without this limitation. Additionally, there are several advances intended for ease of use present in the MUT that do not exist

in the algorithm presented in Fruchterman and Reingold. The MUT automatically processes mesh files of the VTK file format without requiring any conversion. This is useful as the VTK format is a common and open source format for computer graphics and modelling. The MUT does not currently accept other file types. Additionally, the MUT generates data files containing the adjacency matrix and vertex descriptions that can be used by a program such as MATLAB to graph the untangled mesh.

The MUT has multiple operational modes intended to handle various cases of meshes that may commonly be encountered or that are important to untangle. This is another advancement over the Fruchterman-Reingold algorithm which is designed only for a general case and does not deal well with many types of meshes.

3.2: Challenges

3.2.1: Mesh rotation and deformation

One of the issues encountered in the early versions of the MUT was the concept of mesh rotation and deformation using the standard Fruchterman-Reingold algorithm on certain types of meshes. In applications where the original shape and rotation of the mesh are significant, the Fruchterman-Reingold algorithm struggles to produce acceptable results. This issue is most pronounced in meshes with square frames comprised of multiple edges per side. Due to the nature of the attractive forces, these sides easily became bent inwards away from the corners of the original frame, producing convex sides instead of flat ones. This has the effect of pulling the vertices away from the

four corners of the bounding box. Because for a variety of meshes, these four points serves to anchor the mesh, this can be a problem.

In addition, due to the non-fixed nature of mesh frames in certain untangling attempts, the resulting untangled mesh could become rotated as a result of the Fruchterman-Reingold process. Again, this can cause some issues in situations where the position of the mesh in relation to other surfaces is significant. It is generally impossible or at least prohibitively difficult and time consuming to determine the degree of rotation a mesh will undergo in the untangling process mathematically, and the amount of rotation can vary from mesh to mesh. As a result it is not sufficient to simply rotate the mesh by a set amount. This also does nothing to flatten sides that have become convex.

This challenge led to the rise of the Corners-Attract mode. It removes the deformation issue by essentially stretching the corner vertices into the corners and allowing the rest of the graph to adjust to this. Because the Fruchterman-Reingold algorithm seeks to equalize edge lengths, this also has the effect of fixing the rotation issue. When the graph is rotated, fixing the corner vertices causes some edges to become longer as the vertex is pulled away, while other edges become shorter as the vertex moves relatively closer to them. In order to amend this while maintaining these vertices in the corners, the entire graph is forced to shift.

3.2.2: Irregular Frames

Arguably the most significant and influential problem faced in the development of the MUT was how to untangle a mesh with an irregularly shaped boundary in such a way that the boundary is preserved. The defining example of this is the mesh in the shape of the state of Texas, which can be

seen in section 4.1.4. While the Fruchterman-Reingold algorithm is effective on meshes with convex boundaries, it is difficult to adapt to graphs with concave sides and internal holes. Some research has been done in [8] on the subject of convex graphs, but this is of little benefit in the subject of mesh untangling because [8] provides little more than a formal definition, and it certainly does not suggest a way to convert a concave structure into a convex one. Even more pressing is the fact that is generally not advantageous to distort the frame of a mesh during untangling as frame vertex positions should be preserved. Naturally, concave and irregularly framed meshes are rather common, and it is therefore quite important to be able to untangle these despite the challenges they present.

The most defining issue with irregular frames is that it is very difficult to maintain the shape of the frame through the untangling process while still utilizing the Fruchterman-Reingold algorithm. While it is possible to convert most tangled meshes into amorphous untangled cloud-like shapes, this is not desirable in many cases. If the frame of the mesh is distorted when it becomes tangled, without knowledge of the original positions of the mesh vertices, it is nearly impossible to regain the original shape of the mesh. However, if only the internal vertices are displaced when the mesh becomes tangled, opportunities arise.

Another issue stemming from irregular frames is that vertices internal to the frame may become displaced to the outside of the frame, particularly when concave sides are involved. These vertices need to somehow be moved across the frame of the mesh in order to untangle it. Several strategies were tested for this.

First, an attempt was made to simply place every non-frame vertex into the same location internal to the frame. This approach was problematic in a number of ways. In the very first tests, it was determined that the Fruchterman-Reingold algorithm was not equipped to handle the event that two vertices were located in precisely the same location. This resulted in division-by-zero errors in the calculation of attractive and repulsive forces. This minor obstacle was mitigated in the MUT by modifying the algorithm to treat vertices located at the same coordinates as slightly apart, resulting in a powerful repulsive force. However, this was not sufficient to make this method totally reliable, sometimes leading to oddly tangled components or similar bugs.

The second attempt to resolve the issue of irregular frames followed a similar line of thinking to the first, this time only placing the external vertices to an internal location in the frame. This required more complex calculations to determine which vertices were external to the frame. To be more specific, for every edge, calculations had to be performed against every frame edge to determine if they intersected. Not only was this computationally expensive, it was little more effective than the previous method.

The third method to resolve the problem of untangling irregular frames at first met with many obstacles. Conceptually, it was simple. Rather than attempt to untangle the entire mesh at one time, the mesh could be untangled starting from the vertices adjacent to the frame based only on the frame vertices. Due to the stability of the frame structure, it should be possible to move the vertices of just the first layer approximately into their final positions. However, through a series of tests using different levels of attraction and

repulsion as well as varying repetitions for each stage, the results were not as expected.

It was at this point that several issues were discovered. Due to the fact that the MUT was based on the Fruchterman-Reingold algorithm, it possessed several vestigial elements from the original algorithm. Initially, the MUT contained the cooling code possessed by the original, as well as the logic used to prevent vertices from leaving the square bounding box used in the base algorithm. Both of these factors actually negatively impacted the performance of the MUT, especially as the nature of the code changed, and these vestigial sections of logic began to actively interfere with the output. Fortunately, once these were removed, the Fixed-Frame mode was produced which is adept at untangling meshes with irregular boundaries.

3.3: MUT Operation Modes

The MUT has three distinct modes of operation designed to best deal with a variety of different types of meshes. The modes of the MUT are the General mode, the Fixed-Frame mode, and the Corners-Attract mode. The most generally useful mode is Fixed-Frame, with the limitation that the static vertices have to defined as such within the VTK file. Additionally, both the General and the Corners-Attract modes require more room to expand due to their nature. This means that while they may be sufficient for an isolated mesh, the output of these modes may need to be scaled down to fit the original space they took up.

Essentially, the various modes of operation of the MUT provide multiple methods for untangling any given mesh. If the General mode utilizing the Fruchterman-Reingold method is not sufficient and there is present a fixed frame,

one can employ the method of the stage-based untangling presented by the Fixed-Frame mode. In the absence of a fixed frame or presence of a defined square bounding box, the methodology of the Corners-Attract mode provides another possible method of untangling the mesh.

3.3.1: General mode

The General mode of operation is the most basic function of the MUT, and is mostly closely related to the Fruchterman-Reingold algorithm as presented by the authors. The only major difference, as mentioned before, is that the MUT does not employ the concept of temperature, and instead limits the spread of the graph by applying a greater level of internodal attraction. The result of running the MUT in the General mode is usually a more dispersed graph with a roughly circular shape as the structure of the graph allows and relatively evenly distributed vertices. Due to the lack of other restrictions, the mesh spreads out and edge lengths become as close to equal as possible.

The General mode has difficulty untangling meshes with highly unusual shapes. In particular it struggles to produce reasonable untangled solutions for meshes with concave sides and internal holes. It does well with meshes that have a generally vacuous shape or for meshes for which untangling is more important than form; that is to say that is acceptable for the untangled mesh to not resemble the original mesh at all.

3.3.2: Fixed-Frame mode

The arguably most useful mode of MUT operation is the Fixed-Frame mode. It is the most advanced form of mesh untangling implemented in the MUT, and uses a version of the Fruchterman-Reingold algorithm that operates in stages. In the VTK file for a fixed-frame mesh, the vertices that comprise a

static, immobile frame are specified. The code takes the set of vertices directly adjacent to the frame vertices as the first set of adjacent vertices. The Fruchterman-Reingold algorithm is executed as normal over the set of all vertices, but only the adjacent vertices are permitted to move at the movement step.

After a fixed number of iterations, the neighboring vertices to the current set are selected as the next set of adjacent vertices, and the current set of adjacent vertices is added to the set of processed vertices. During the move step, the set of processed vertices are allowed to move, but at a greatly reduced rate to avoid the impairment of properly untangled vertices while still allowing for improvement. The process repeats several times, after which the remaining vertices are all considered to be the next set of adjacent vertices and are untangled as described above.

The major weakness of the Fixed-Frame mode of operation is that the frame vertices must be specified in the VTK, and not all meshes may be compatible as a result. However, the performance of the Fixed-Frame mode generally far outweighs this limitation. As will be demonstrated in the experimental results, the Fixed Frame mode accurately untangles meshes of very unusual shapes, including meshes with concave sides, internal holes, and others.

3.3.3: Corners-Attract mode

The third and final operational mode of the MUT is the Corners-Attract mode. It is a highly specialized mode to deal with a particular special case of tangled mesh. In the event that a mesh is not of the Fixed-Frame variety, but the mesh fits in a rectangular frame such that there is a vertex in each of the

corners, a slightly modified application of the standard Fruchterman-Reingold algorithm can produce satisfactory results. The Corners-Attract mode operates identically to the General mode for a given number of iterations to produce an initial untangling. However, due to the untangling process this variety of square mesh is often twisted or rotated from the original, in addition to deforming from the overall square shape. To compensate for this, the Corners-Attract mode heuristically determines the proper vertices to affix to the corners and forcibly moves them there. The Fruchterman-Reingold algorithm is repeated on the mesh for a number of iterations with the corner vertices in the corners, causing the mesh to conform to the rectangular frame.

The main weakness of the Corners-Attract mode is that it applies only to specific cases of meshes. However, it can be used for general case meshes if the user desires the output mesh be square. The advantage of the Corners-Attract mode is primarily its ability to untangle rectangular meshes in the absence of a fixed frame.

3.4: Psuedocode

Below is the final version of the psuedocode of the algorithm developed. The implementation of the final algorithm in the Java programming language can be found in appendix (A).

Given:

-G(V,E) is the input graph

-xmax/xmin and ymax/ymin describe the location of the frame's corners in a Corners-Attract style mesh and are also used to determine the value of k

-An edge E is an ordered pair of vertices v and u

-F is the list of fixed frame vertices in a Fixed-Frame type mesh

-A is the list of adjacent vertices in a Fixed-Frame type mesh

-P is the list of proximal vertices in a Fixed-Frame type mesh

$width := xmax - xmin$

$height := ymax - ymin$

$area := width \times height$

$k := \sqrt{area/|V|}$

for i from 1 to $iterations$ begin

{ repulsive forces }

for v in V begin

$v.disp := 0$

for u in V begin

if $u \neq v$

$\Delta := v.pos - u.pos$

if $|\Delta| == 0$

{ treat vertices with same position as slightly apart }

$|\Delta| :=.001$

$v.disp += \left(\Delta/|\Delta|\right) \times \left(k^2/|\Delta|\right)$

end

end

{ attractive forces }

for e in E begin

$\Delta := e.v.pos - e.u.pos$

if $|\Delta| == 0$

{ treat vertices with same position as slightly apart }

$|\Delta| :=.001$

$v.disp += \left(\Delta/|\Delta|\right) \times \left(|\Delta|^2/k\right)$

$$u.disp -= \left({}^{\Delta}/_{|\Delta|}\right) \times \left({}^{|\Delta|^2}/_{k}\right)$$

end

{ move vertices }

for v in V begin

$d := |v.disp|$

{ for Fixed-Frame type mesh, reduce movement for processed vertices }

if $type == Fixed\text{-}Frame$ and $v \in P$

$d := d \times 10$

{ for Fixed-Frame type mesh, don't move vertex if not frame or adjacent}

if $type \neq Fixed\text{-}Frame$ or $(\neg v \in F$ and $v \in A)$

$$v.pos += \left({}^{v.disp}/_{d}\right)$$

End

3.5: Mathematic Quantification of Untangling

The concept of mesh untangling is obviously very important to this thesis, but the mathematically definition of an untangled mesh has not been discussed. It is relatively simple to assess the status of a mesh visually, as it can be determined whether there are edge crossings and deformation by viewing the plot the mesh is Cartesian space. However, this is not always an option, and in the case of an active system it may be necessary to be able to determine if a mesh is untangled mathematically.

Fortunately, it is possible to determine mathematically if a mesh is untangled, and even the degree of untangling of the mesh is based on several factors. The most basic metric for mesh quality is inversion of components. In a triangular mesh, a triangular component is said to be inverted if the volume of the

component is negative. If a mesh contains inverted components, then it can be said to be tangled.

The case of polygonal meshes is essentially the same. For quadrilateral meshes, if any of the quadrilaterals are inverted then the mesh is considered to be tangled. This is done slightly differently, however. There are four triangles that can be formed out of the vertices for a quadrilateral, and each one is checked for inversion in the same way the components of triangular mesh are. If any of these triangles are inverted, it can be determined that the quad itself is inverted.

The case of hybrid meshes with components of multiple varieties is very straightforward. Put simply, the test appropriate to that component is run for each component to determine if it is inverted, and thus if the mesh is tangled. It is also possible to determine the approximate level of tangling by comparing the number of tangled components to the overall number of components in the mesh.

4. Experimental Results

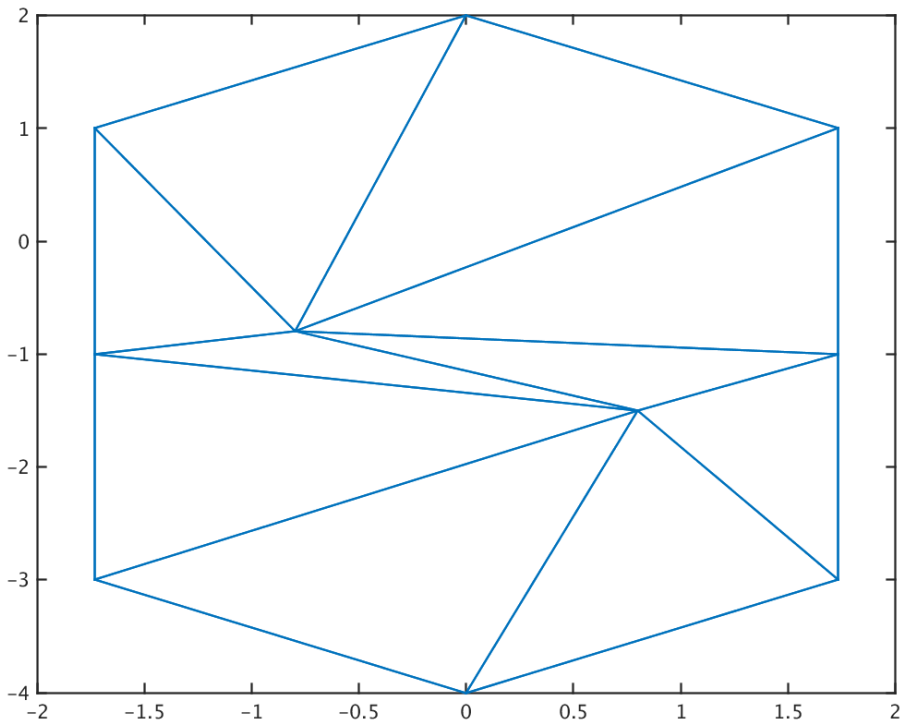
The following consists of examples of the results of running the MUT on various types of meshes. All operational modes are demonstrated to illustrate the effectiveness of the various modes on disparate mesh types. The quantification of how tangled various results are based on the concepts in section 3.5 is presented along with the graphs in the form of the graph's tangling number which represents the number of components that are inverted. The initial tangling number of the tangled graph before it is processed will be given, but the tangling number for an untangled graph will be omitted as it is always 0.

4.1: Triangular Meshes

The most basic mesh structure is that of the triangular mesh. The components of a triangular mesh are all triangles, which are stable structures from a geometric perspective. The MUT has very good success with the untangling of triangular meshes, several examples of which will be shown below.

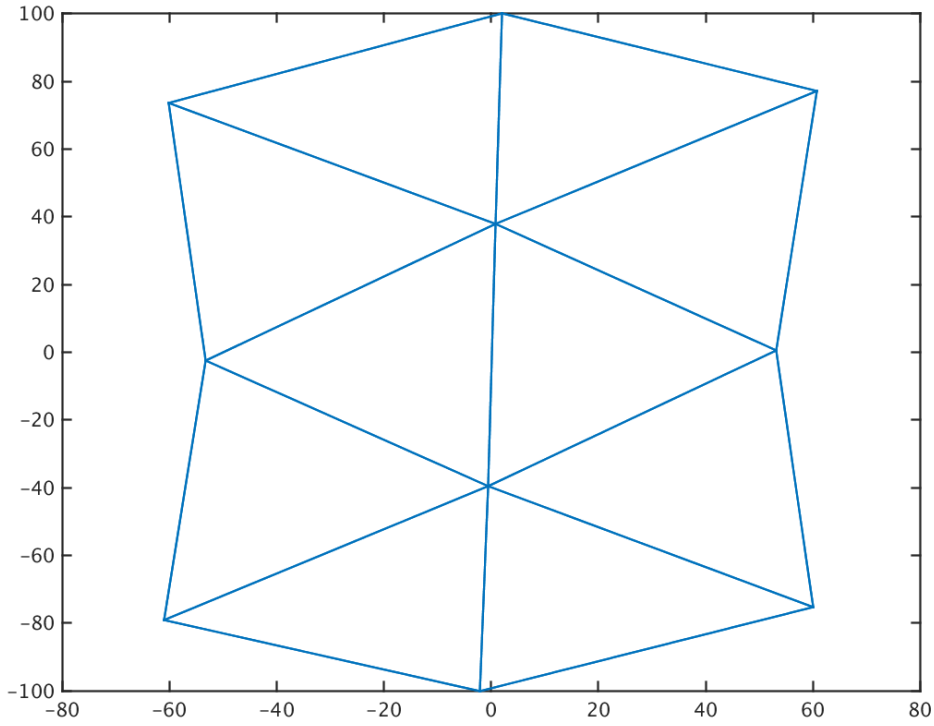
4.1.1: Example Mesh 1

A relatively humble and simple mesh, Example Mesh 1 was the first mesh that the MUT was tested on in the prototype stages. Tangled, it is shown in Figure 1. The tangling number is 0, because while this mesh is certainly distorted it is not genuinely tangled. The purpose of this mesh was to test the functionality of the Fruchterman-Reingold algorithm on simple meshes.



(Figure 1: Example Mesh 1 – Tangled)

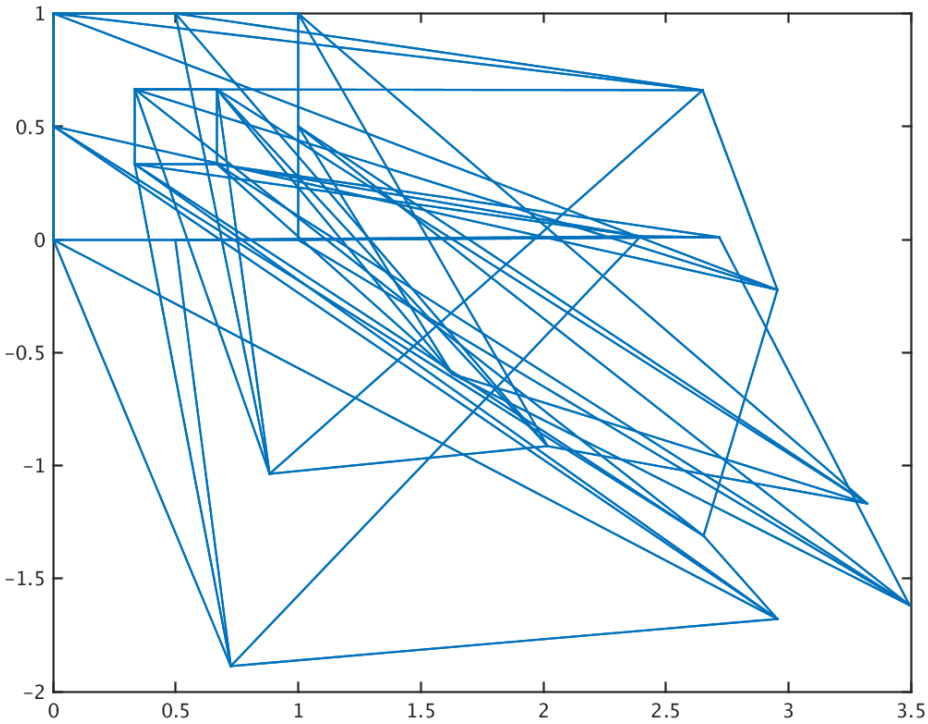
It does not have a fixed frame, so that mode of operation produces no results. Additionally, the result of running the Corners-Attract mode on this mesh has been omitted. Example Mesh 1 is a good example of the type of mesh that the base Fruchterman-Reingold algorithm performs well on. Figure 2 shows the mesh untangled with the General mode of operation.



(Figure 2: Example Mesh 1 – General)

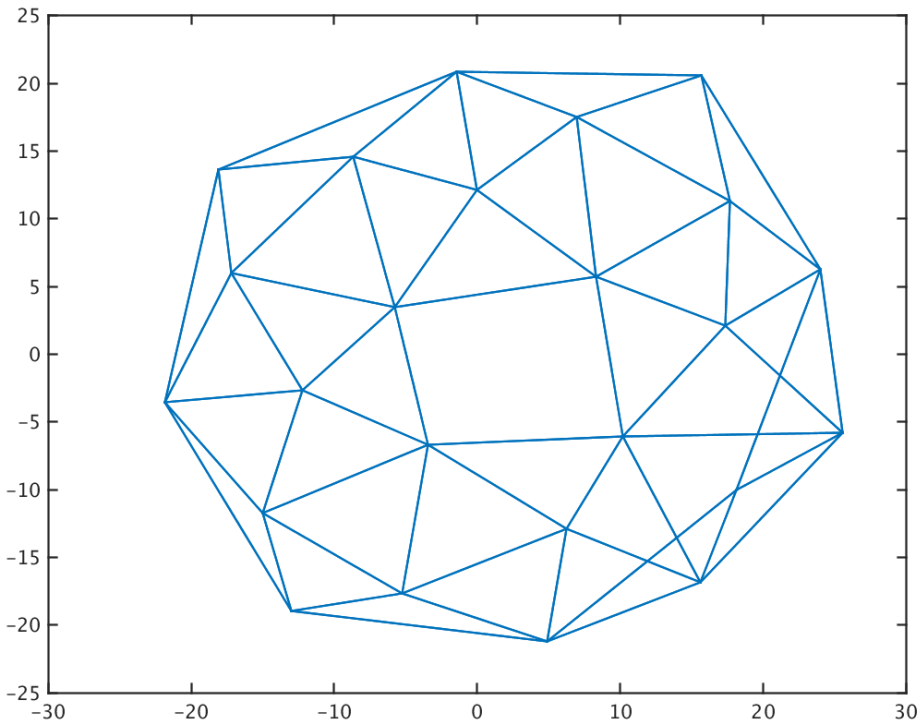
4.1.2: Square Mesh with Square Hole

This mesh is another simple triangular mesh. It has both a square frame when untangled and an internal hole, also in the shape of a square. Tangled, it is shown in Figure 3. It has a tangling number of 16.



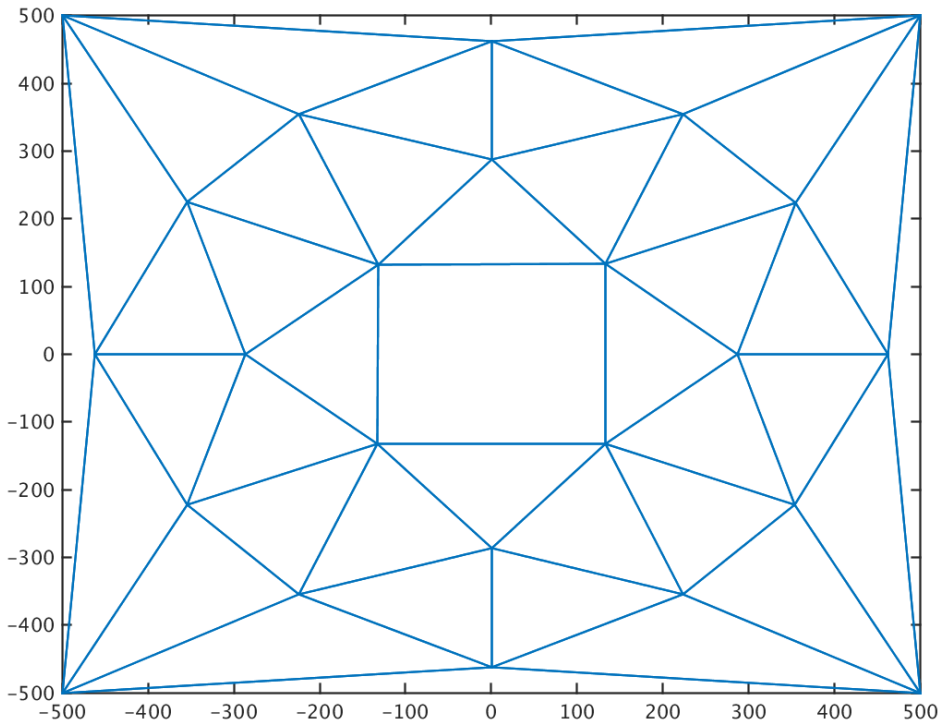
(Figure 3: Square Mesh with Square Hole – Tangled)

The General operation mode meets with near success for this mesh. In order to allow enough room for expansion, the bounding box given to the mesh was 1000 by 1000 units. Despite this large space, the attractive forces between vertices restrain the expansion of the mesh as seen in Figure 4 below. Unfortunately, there is some tangling still present in the lower right hand corner of the mesh. The tangling number is 3.



(Figure 4: Square Mesh with Square Hole – General)

The Corners-Attract mode functions admirably for this mesh. With the parameters properly assigned, the MUT correctly identifies the four corner vertices and untangles the graph as shown in Figure 5.

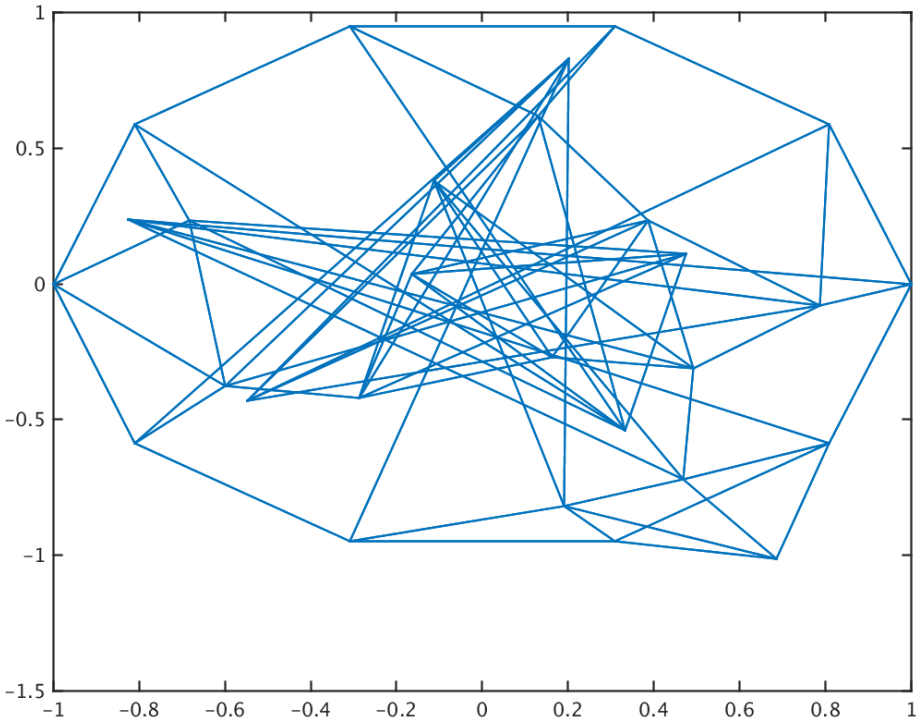


(Figure 5: Square Mesh with Square Hole – Corners-Attract)

While this does not completely overcome the distortion produced by tangling, it effectively untangles the mesh and most significantly preserves the four corners of the original mesh. As there are no frame vertices specified in the file, the Fixed-Frame mode cannot process this mesh.

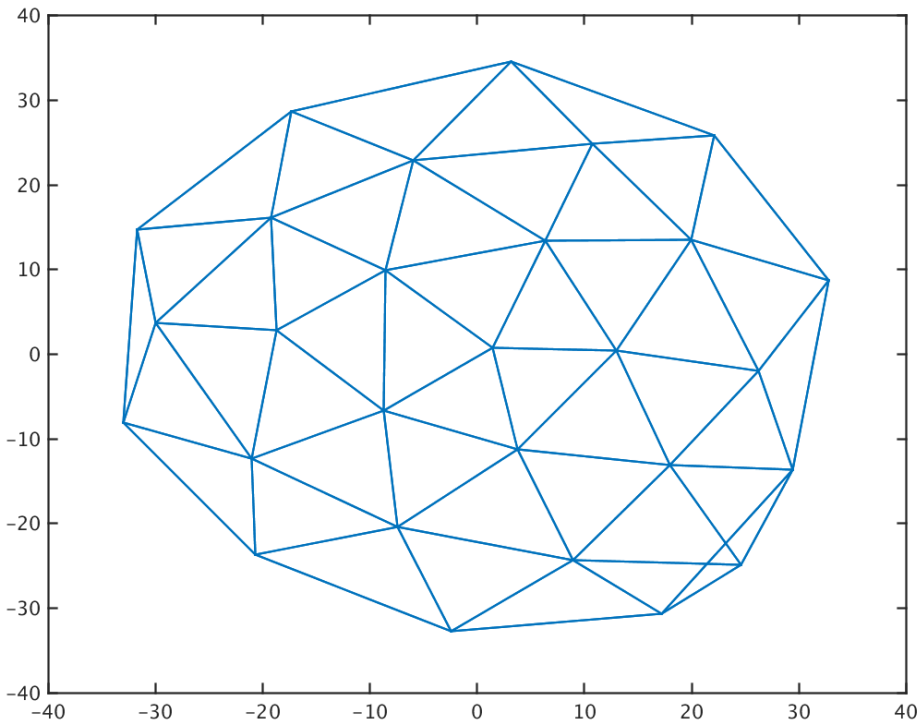
4.1.3: Disk

The Disk mesh is slightly more complicated of a mesh than those viewed previously. Untangled, it consists of a “disk” formed by a large outer polygon with the rest of the vertices within this frame. This mesh is the first example of a mesh with a fixed frame that we will discuss. While still tangled, it is shown in Figure 6. It has a tangling number of 15



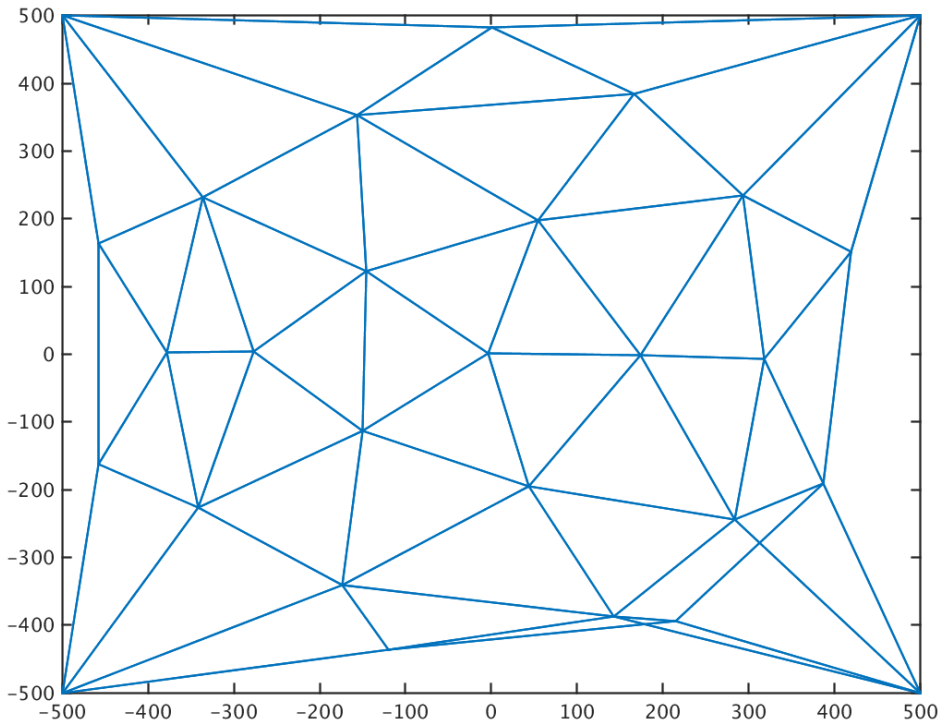
(Figure 6: Disk – Tangled)

The General mode of operation comes quite close to a proper untangling, as shown in Figure 7 below. However, there is still tangling present in the bottom-right section of the graph. As there is one inverted component, the tangling number is 1.



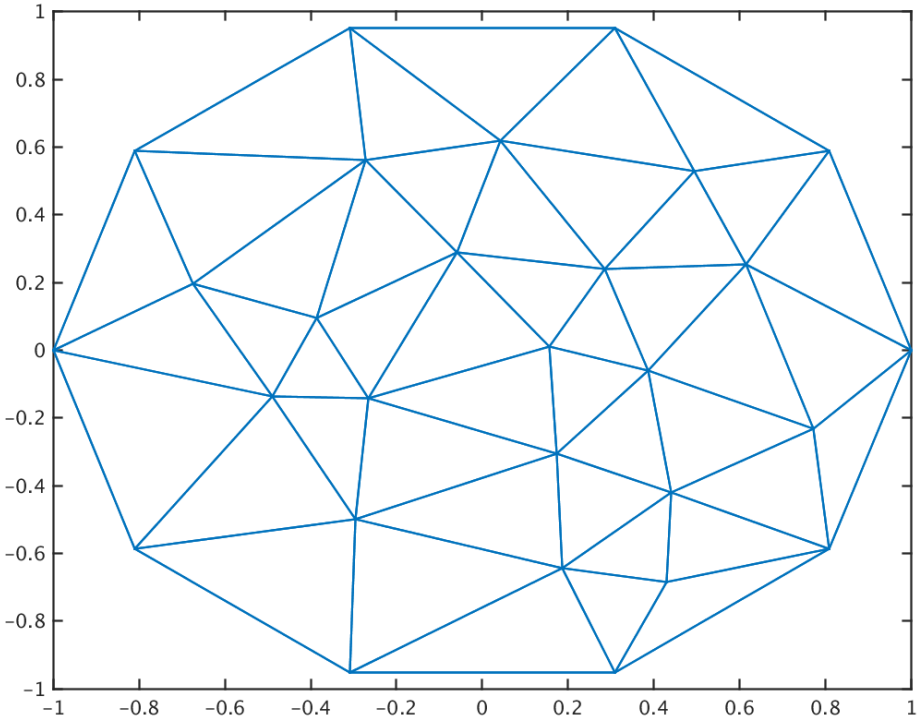
(Figure 7: Disk – General)

The Corners-Attract mode falls short for the Disk mesh due to misidentification of the corner vertices, which is understandable considering the mesh had none to begin with. This leads to a mesh that is still tangled as in Figure 8. The tangling number for this graph is 2.



(Figure 8: Disk – Corners-Attract)

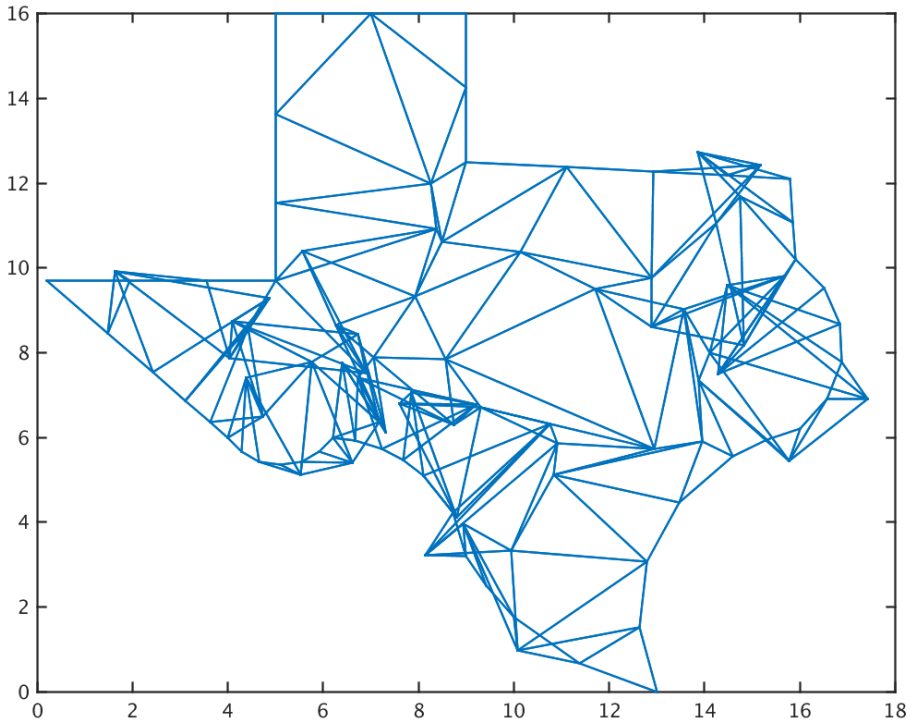
As mentioned prior, the Disk mesh has a frame that is fixed and identified within the original file. These vertices are not moved during the tangling process and can be used to anchor the rest of the untangling effort when using the Fixed-Frame mode. This mode is successful in untangling the mesh as shown in Figure 9.



(Figure 9: Disk – Fixed-Frame)

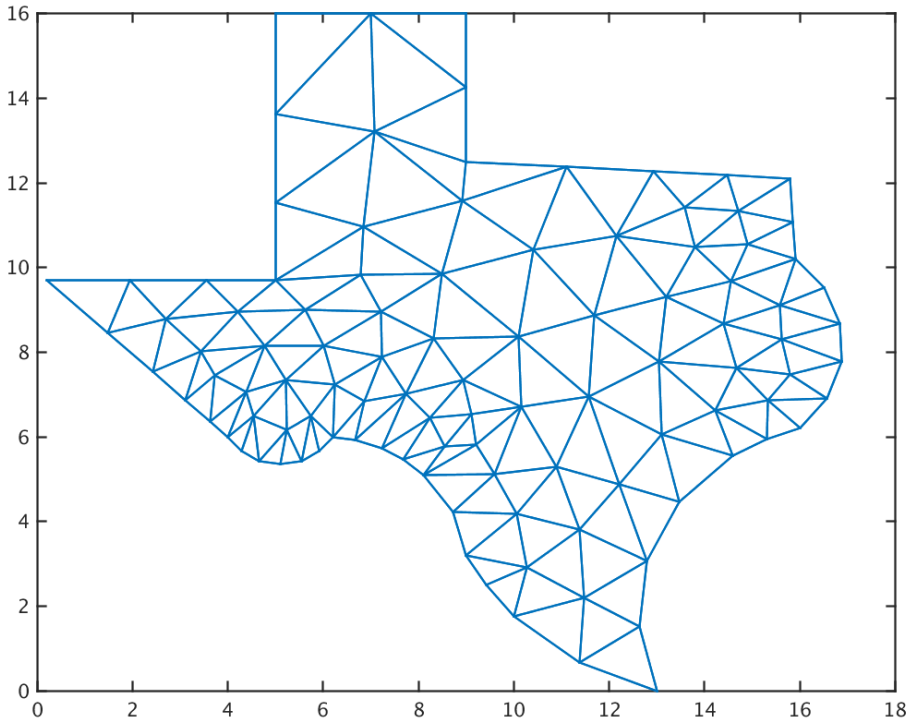
4.1.4: Texas

The Texas mesh was the inspiring example for the MUT. When the problem was proposed, it was in the following terms: “How would you untangle a mesh with an irregular border, for example, the state of Texas?” This mesh inspired the creation of the Fixed-Frame mode, and not surprisingly only shows results under that treatment. As the shape of the mesh is of paramount importance, only the tangled mesh and the untangled mesh using the Fixed-Frame mode are presented. The tangled mesh is as shown below in Figure 10. The tangling number of the tangled mesh is 46.



(Figure 10: Texas – Tangled)

The successfully untangled mesh using the Fixed-Frame mode is shown below in Figure 11.



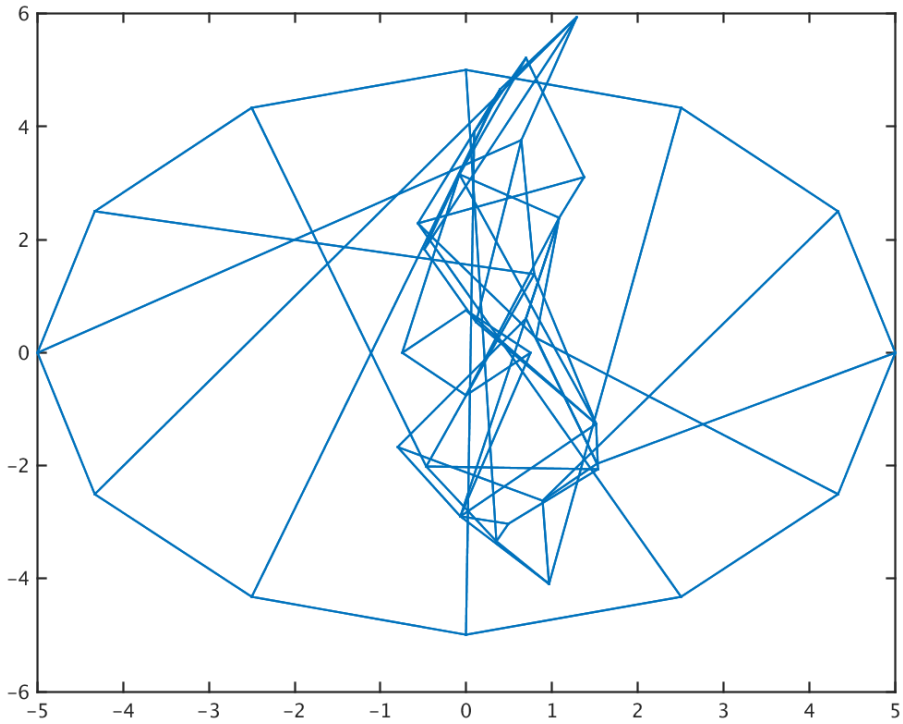
(Figure 11: Texas – Fixed-Frame)

4.2: Quadrilateral Meshes

More complicated than triangular meshes are quadrilateral or quad meshes. Fundamentally they are very similar to triangular meshes, but with quadrilateral components. This makes them somewhat more complicated to untangle because the triangle is a very geometrically stable structure, and the quadrilateral is much less so.

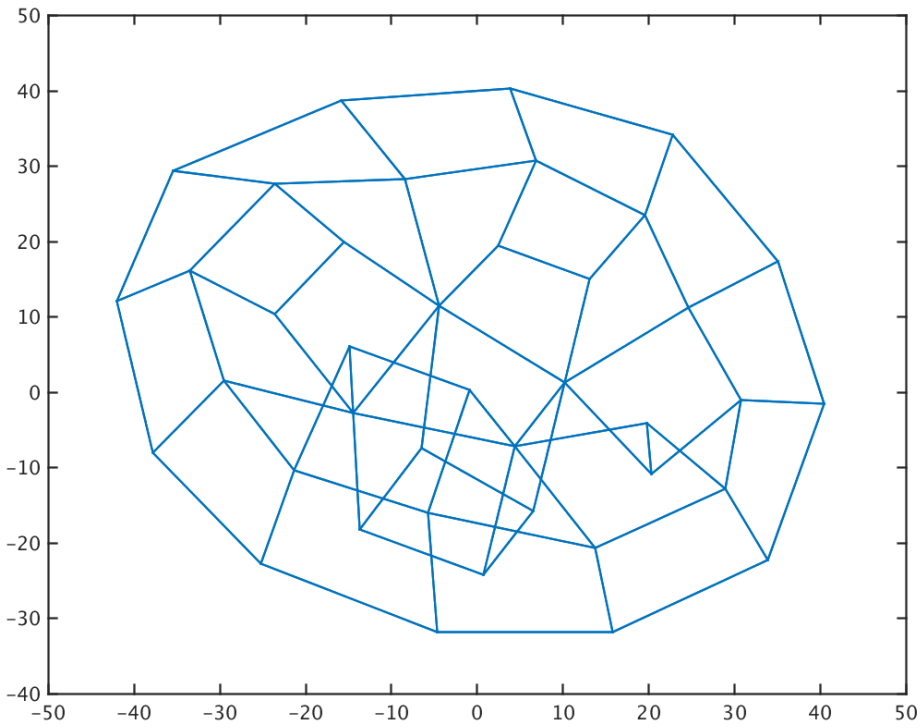
4.2.1: Eye of the Tiger Quad

The so-called Eye of the Tiger Quad mesh gets its name from its original shape, consisting of a spiral of quads with a rhomboid hole in the center. The tangled version of this mesh is shown in Figure 12 below. The tangling number is 19.



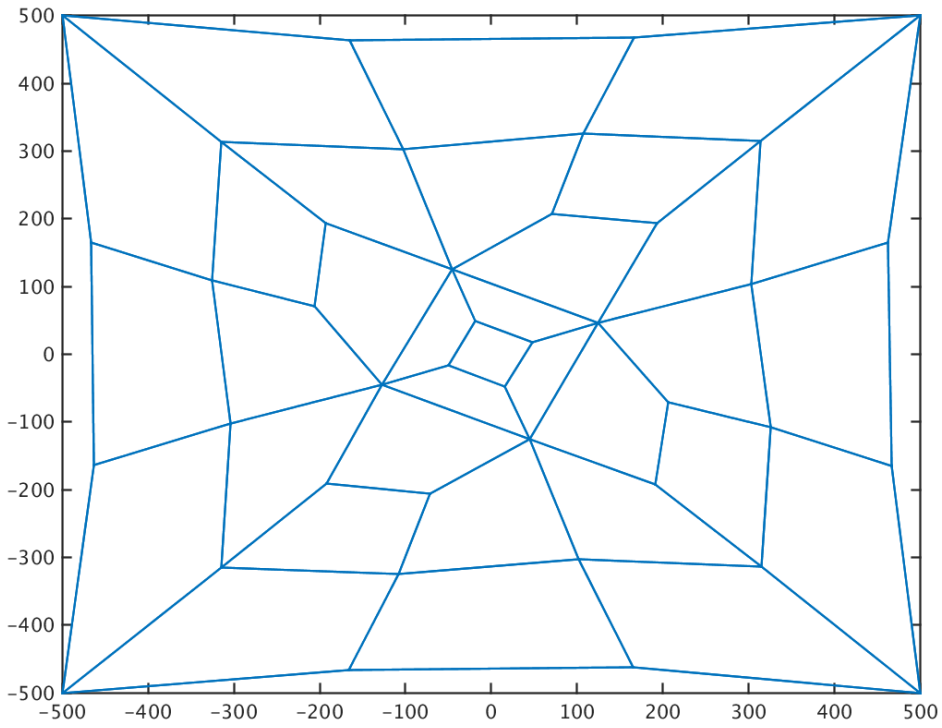
(Figure 12: Eye of the Tiger Quad – Tangled)

The General mode struggles greatly with quad meshes. The output of the General mode on this mesh is still very tangled as shown in Figure 13. The tangling number is 6.



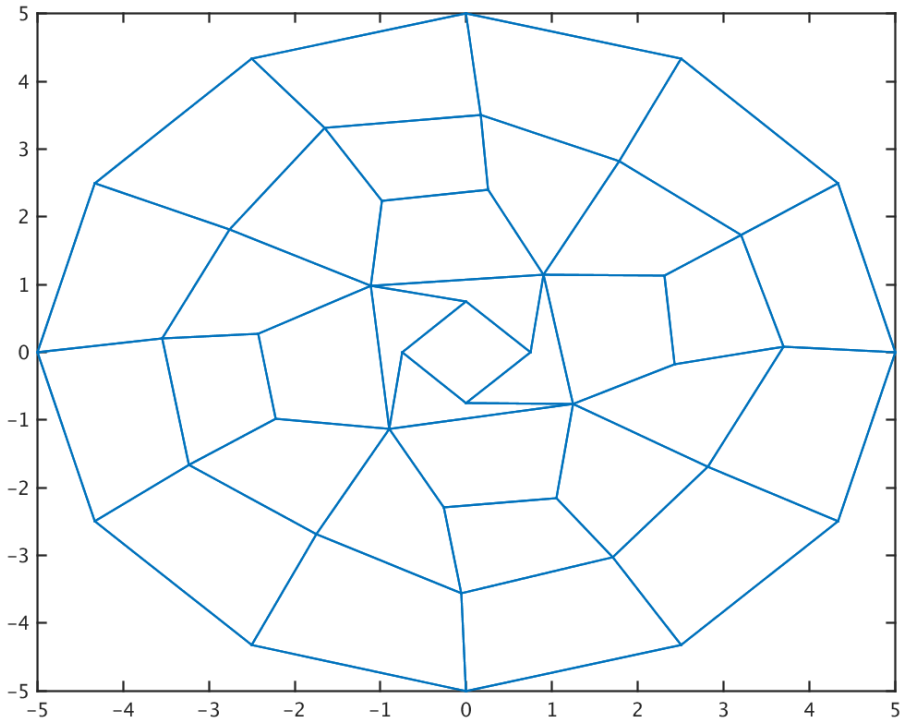
(Figure 13: Eye of the Tiger Quad – General)

The Corners-Attract mode works surprisingly well with the Eye of the Tiger Quad mesh despite the fact that the original shape is not a square. However, the correct corner vertices were identified by the MUT, and the output is shown in Figure 14.



(Figure 14: Eye of the Tiger Quad – Corners-Attract)

The Fixed-Frame mode is also effective at untangling the mesh, and preserves more of the original shape as shown in Figure 15.

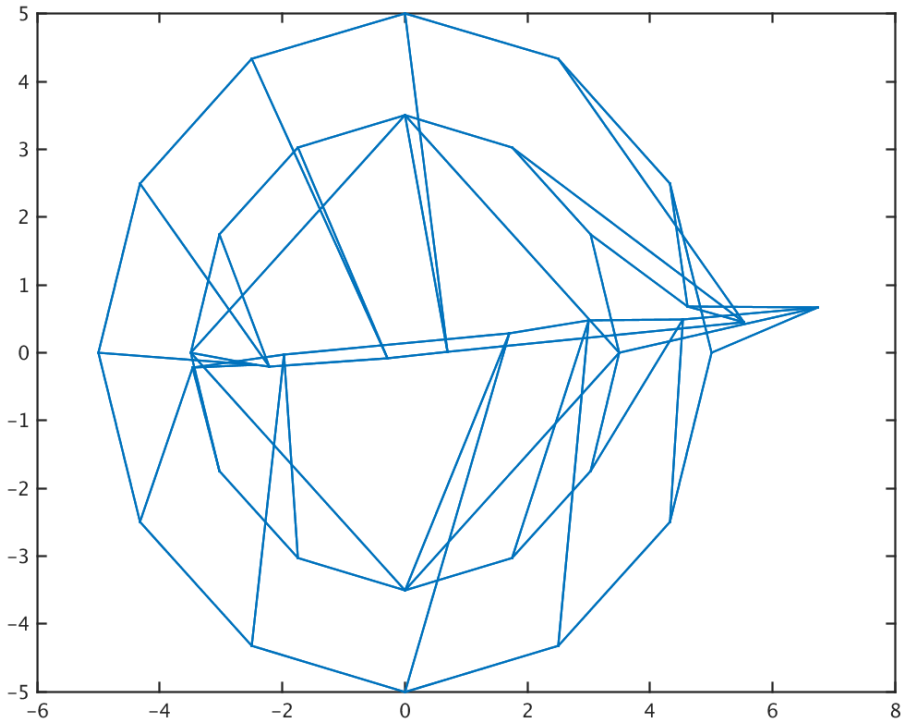


(Figure 15: Eye of the Tiger Quad – Fixed-Frame)

From an aesthetic standpoint it is difficult to determine which of the two successful untangled meshes is preferable. However, in a practical application the Fixed-Frame mode would likely be superior as it has preserved the original frame of the mesh.

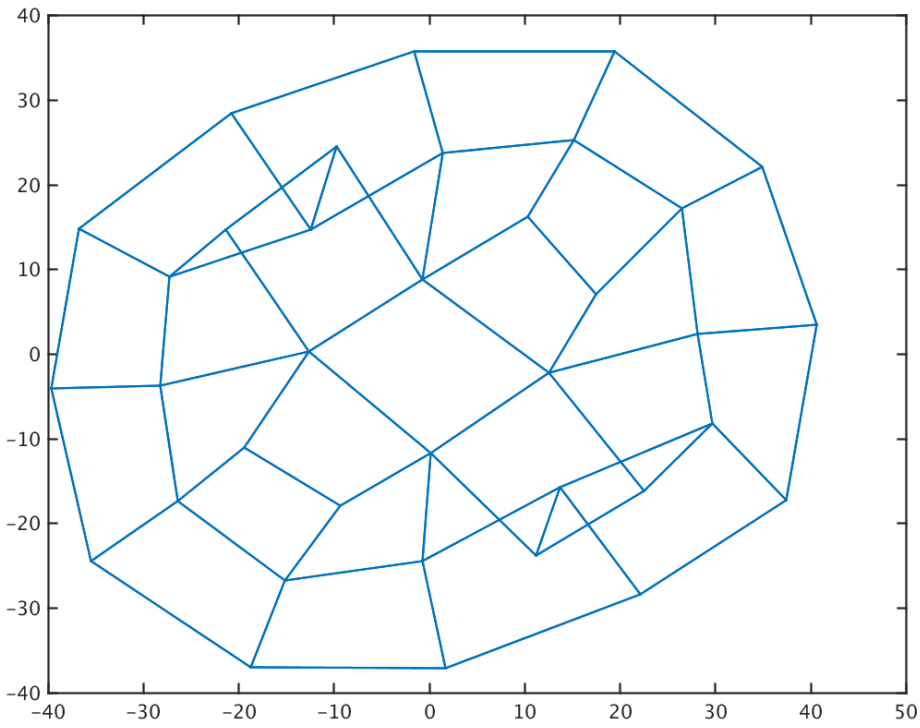
4.2.2: Annulus Quad

The Annulus mesh consists of a rhomboid hole in the center of a polygonal mesh. In the original mesh, the square hole is very large compared to the size of the mesh. The tangled mesh is shown below in Figure 16. It has a tangling number of 12.



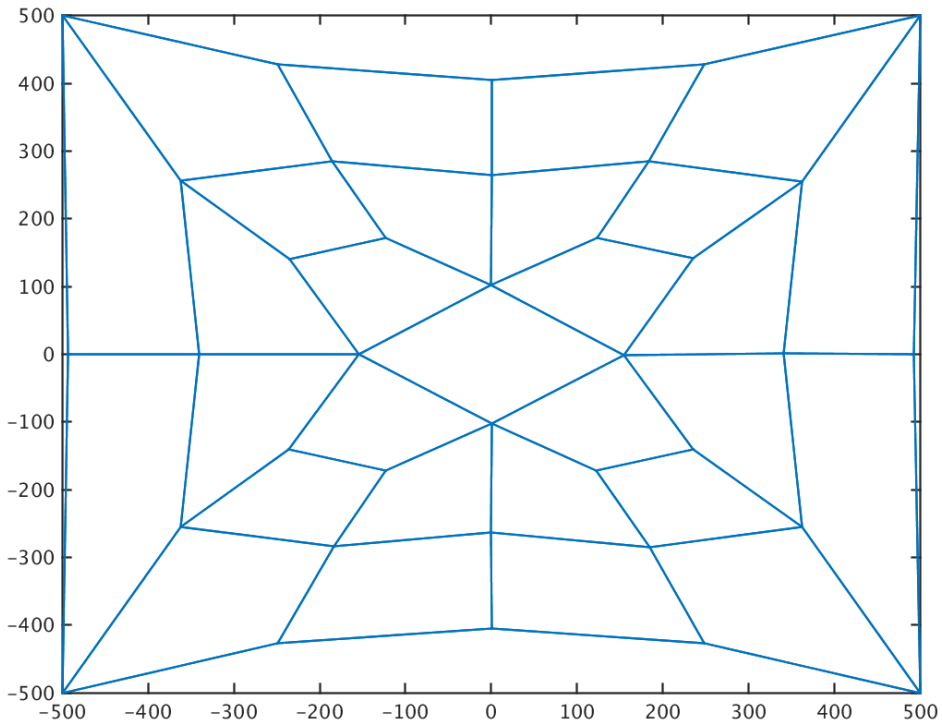
(Figure 16: Annulus Quad – Tangled)

The General mode comes surprisingly close to untangling this mesh but there are still several areas of tangling as shown in Figure 17. The tangling number is 4.



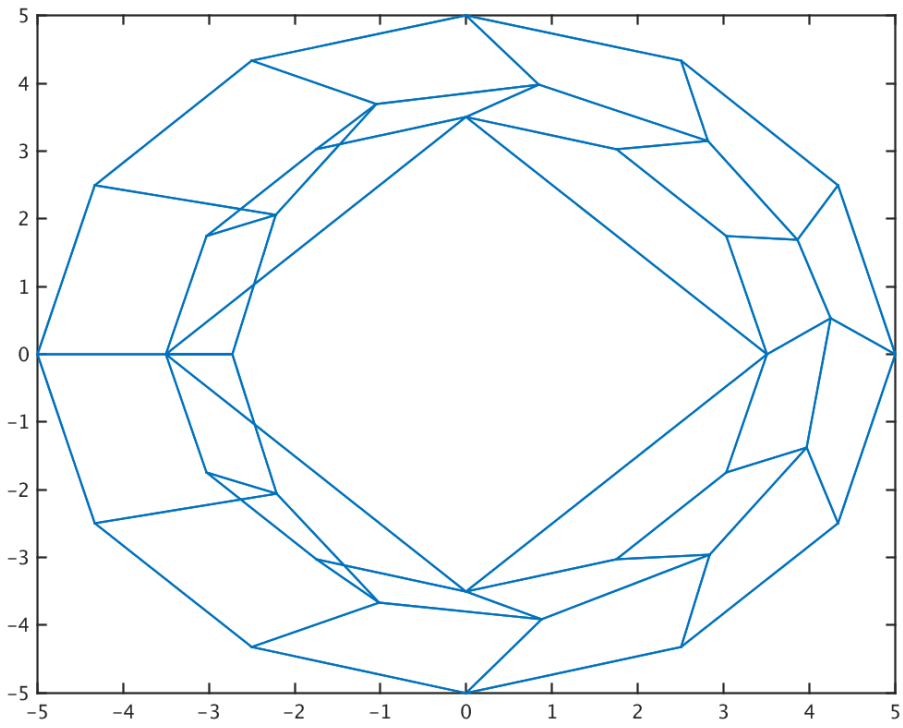
(Figure 17: Annulus Quad – General)

Much like the Eye of the Tiger Quad mesh, the Annulus Quad mesh also responds relatively well to the Corners-Attract mode, untangling though it loses its shape as in Figure 18.



(Figure 18: Annulus Quad – Corners-Attract)

Perhaps the most surprising thing about the Annulus Quad mesh is its inability to be untangled with the Fixed-Frame mode. Despite trying many different values to increase and decrease the force of attraction between vertices, it was not possible to determine how to untangle the mesh with this mode in a reasonable amount of time, making it of little use in an active simulation. One such attempt is shown below in Figure 19. The tangling number for this graph is 4, proving that it is no more effective than the General mode.



(Figure 19: Annulus Quad – Fixed-Frame)

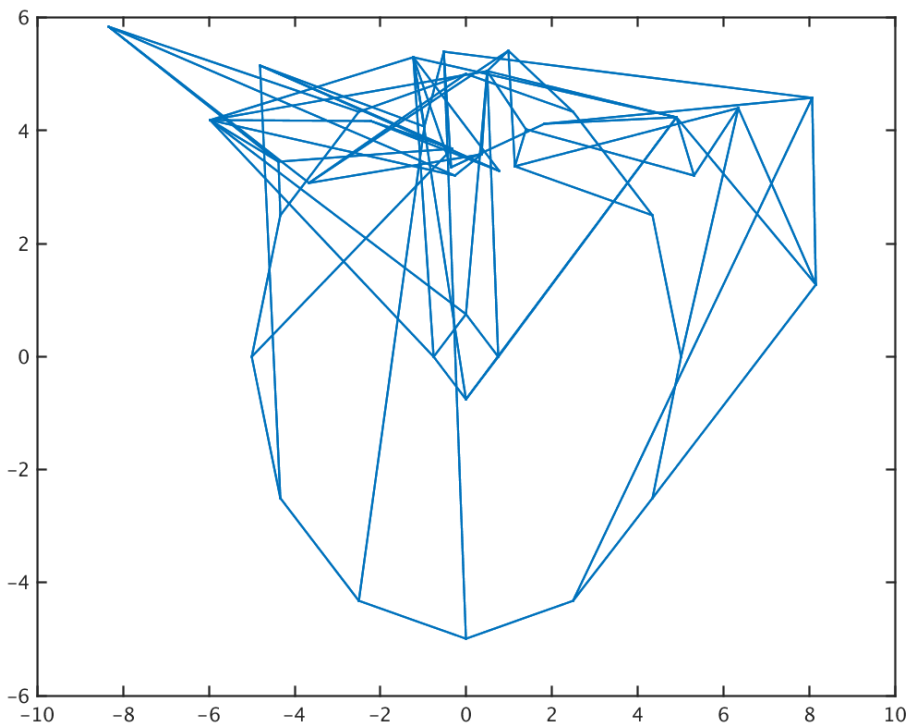
It is likely that the issues the MUT has in untangling this mesh stem from the relative size of the internal hole compared to the size of the mesh as a whole. Since the Fruchterman-Reingold algorithm attempts to approximately equalize edge lengths, the very long, fixed edges of the internal hole make the algorithm fail. Since the edge lengths are not fixed in the Corners-Attract mode, and the mesh is stretched a great deal, this is not a problem. However, as discussed before, the output of the Corners-Attract mode may not be very useful given that the original shape is lost.

4.3: Hybrid Meshes

Hybrid Meshes consist of components of multiple varieties. Here we will examine a hybrid mesh consisting of quads and triangles.

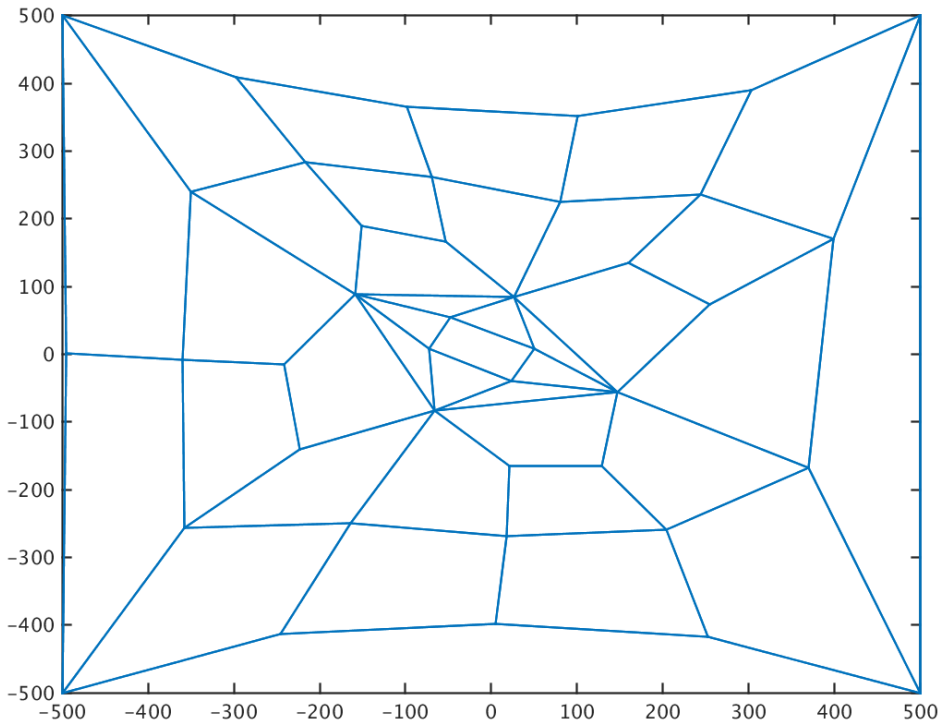
4.3.1: Eye of the Tiger Hybrid

The Eye of the Tiger Hybrid mesh is a simple conversion of the quad version created by converting some of the quads to triangles by connecting a pair of diagonal vertices. The tangled version of the mesh is shown in Figure 20 below.



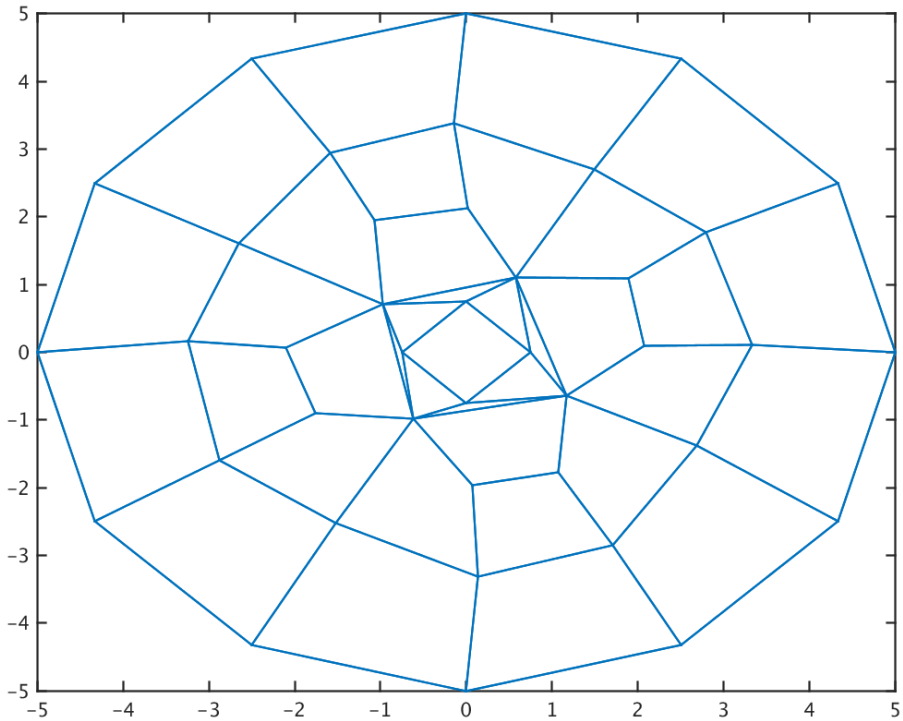
(Figure 20: Eye of the Tiger Hybrid – Tangled)

The General mode produces very similar results to the quad version and will not be shown. Surprisingly, the Corners-Attract mode manages to untangle the mesh, if rather unattractively as in Figure 21.



(Figure 21: Eye of the Tiger Hybrid – Corners-Attract)

The Fixed-Frame mode produces more desirable results as shown in Figure 22.



(Figure 22: Eye of the Tiger Hybrid – Fixed-Frame)

Like the other meshes for which both Corners-Attract and Fixed-Frame modes produce untangled results, it is likely that the latter will be more valuable as it more accurately matches the shape of the original mesh.

5. Conclusions

5.1: Summary

Overall, the MUT (Mesh Untangling Tool) and the methodologies applied in it are very successful at untangling meshes. Utilizing three different operating paradigms, the MUT was able to untangle many complicated meshes that were previously impossible to untangle through a graph embedding approach. It is capable of preserving complex frames with concave elements and internal holes as well as meshes without frames that conform to a rectangular bounding box. The untangling is performed with a high degree of accuracy and some constants can be modified on a graph by graph basis in order to adapt to special cases.

One of the major weaknesses of the MUT is that it cannot presently process graphs in three-dimensional space. Additionally, some cases require large changes to constants or are impossible to accurately untangle using the MUT. These include cases where an internal hole is very large, such as the Annulus Quad mesh. Because the Fruchterman-Reingold algorithm strives to make edge lengths uniform, having a component that is much larger than the others reduces the effectiveness of the method.

5.2: Future Work

The most immediately obvious area of expansion for the MUT and graph embedding-based mesh untangling paradigm is to move into the three-dimensional space. In real-world applications many meshes will be three-dimensional in nature, which limits the usefulness of a solely two-dimensional untangling tool. Theoretically speaking, it should be straightforward to take three dimensions into account. The principles of making edge lengths as uniform as

possible and avoiding edge crossing are both applicable to three-dimensional space as well as two-dimensional.

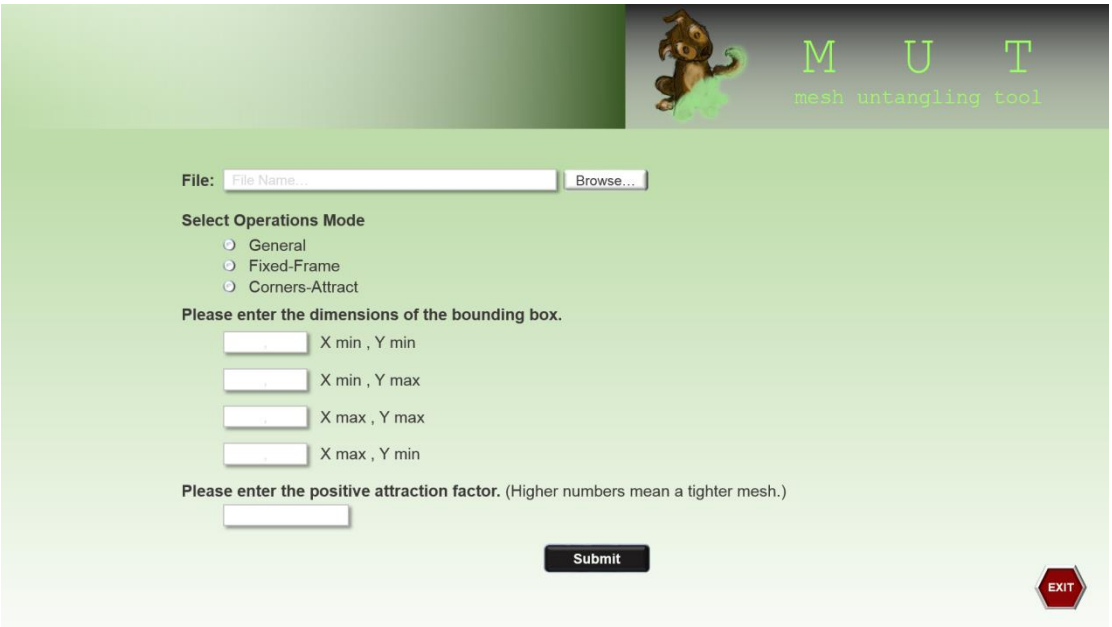
Another area to explore would be a better method of handling extreme meshes, such as the aforementioned Annulus Quad mesh that are currently difficult to handle with the MUT. Introducing a new mode of operation to deal with meshes that have components of unusually large size compared to the other components of the mesh could potentially alleviate this problem. More work has to be done to determine what should be done differently in the algorithm to handle such cases. Lastly, I have not researched the efficiency of the MUT compared to other mesh untangling methods as that is outside the scope of this project. Such research would be essential in order to advocate the use of the methods used in the MUT over other contemporary methods for mesh untangling.

Possibly the most beneficial area of expansion for the MUT would be to produce an attractive and easy to use user interface for the tool. Currently, the program is run from the command line on the mesh files, and the output of the MUT is used by a program such as MATLAB to produce graphs. For a lone researcher, this is acceptable; however one way the tool would be more useful would be if it were easily usable by someone with less programming experience, and less familiarity with the nuances of the code, though the MUT is not currently available online. Nonetheless a short description of how to run the code is given below:

1. Run the MUT from the command line: “./MUT”
2. Provide the file path to VTK file to untangle: “../VTK_files/disk_tangled.vtk”
3. Respond to prompt for the operation mode (all modes take the same parameters)

4. Respond to prompts for bounding box corner coordinates (in the case of fixed frame, minimize bounding box to be no larger than the frame requires)
5. Respond to prompt for attraction factor (1 for standard attraction)

A user interface where the VTK file could be uploaded and all options set up and edited at once would be highly useful. A mock-up of what this could possibly look like is shown in Figure 23 below. The untangled graph in VTK format a picture of it could be returned to the user.



(Figure 23: MUT User Interface Mock-Up)

In line with the idea of a user interface, an even more audacious area of future expansion also comes to mind. The MUT along with many other algorithms and methods of mesh untangling operate independently of user input, at least after initial information. They run as they are programmed to and produce output that hopeful will be untangled. One way to expand the MUT would be to allow users to interact with the untangling process. There are two main ways I imagine this may be useful.

First, in the Corners-Attract mode, sometimes the algorithm determines the wrong vertices to be the corner vertices if the mesh is not sufficiently untangled at the point this is determined. It would be very useful if at that point the user were able to audit the code's selections to correct them. The other place where interaction between the user and the MUT may be useful would be in the Fixed-Frame mode.

In the case of the Fixed-Frame mode, the most logical time for user interaction would likely be in between stages. At that point, the user could make several modifications to the output. First, I think it would be useful for the user to be able to freeze certain vertices as if they were frame vertices at will. This could improve the quality of the untangling by ensuring key non-frame vertices were in the ideal position. Additionally, it may be useful to allow the user to move vertices in order to help overcome very bad placements which the code is unsuccessful at fixing. This could also improve the quality of the output and allow the MUT to successfully untangle meshes it could not before, such as the Annulus quad mesh using the Fixed-Frame mode.

To do this, several things would have to be done. First, it would be necessary to be able to display an image of the mesh in real time. It would also be necessary to map this image of the graph to the data in such a way that the user could modify the mesh through this interaction. Allowing user interaction in the untangling process in this way could provide significant improvements in function for the MUT over the success it has already seen.

References:

[1] Fruchterman, T. M. J., & Reingold, E. M. (1991). Graph Drawing by Force-Directed Placement. Software: Practice and Experience, 21(11).
[2] G. Agnarsson and R. Greenlaw, *Graph theory: Modeling, applications, and algorithms*, Graph Theory: Modeling, Applications, and Algorithms, Pearson/Prentice Hall, 2007
[3] G. Di Battista , P. Eades , R. Tamassia , I. Tollis, “Algorithms for drawing graphs: an annotated bibliography,” Computational Geometry: Theory and Applications, v.4 n.5, p.235-282, 1994.
[4] T. Kamada and S. Kawai, “An Algorithm for Drawing General Undirected Graphs," Information Processing Letters, vol. 31, pp. 7-15, 1989.
[5] D. Tunkelang, “A Practical Approach to Drawing Undirected Graphs," M.S. Thesis, School of Computer Science, Carnegie Mellon, 1994.
[6] H. Harborth and I. Mengersen, “Edges Without Crossings in Drawings of Complete Graphs," J. Combinatorial Theory (B), vol. 17, no. 3, pp. 229-311, 1974.
[7] D. Ferrari and L. Mezzalira, “On Drawing a Graph with the Minimum Number of Crossings," Technical Report n. 69-11, Istituto di Elettrotecnica ed Elettronica, Politecnico di Milano, 1969.
[8] S.K. Stein, “Convex Maps," Proc. Amer. Math. Soc., vol. 2, pp. 464-466, 1951.
[9] T. Ozawa, “Planarity Testing for IC Layout with Constraints for Pin Order and Congestion Between Pins," IEEE Conf. Record of the 14th Asilomar Conf. on Circuits, Systems Computers, pp. 188-192, 1980.
[10] P. Eades, “A Heuristic for Graph Drawing," *Congressus Numerantium*, vol. 42, pp. 149-160, 1984.
[11] R. Lipton, S. North, and J. Sandberg, “A Method for Drawing Graphs," Proc. ACM Symp. on Computational Geometry, pp. 153-160, 1985.
[12] A. Yamaguchi, and H. Toh, “Visualization of Genetic Networks: Edge Crossing Minimization of a Graph Drawing with Vertex Pairs," Genome Informatics 11, pp. 245–246, 2000.
[13] Julia Chuzhoy, “An Algorithm for the Graph Crossing Number Problem,” Proc. ACM Symp. on Theory of Computing, 2011.
[14] J. Brank, “Drawing graphs using simulated annealing and gradient descent,” Department of Knowledge Technologies, Jozef Stefan Institute.
[15] R. Davidson, D. Harel, “Drawing graphs nicely using simulated annealing,” ACM Transactions on Graphics, 15(4):301–331, 1996.
[16] R.K. Guy, “Crossing Numbers of Graphs," Graph Theory and Applications, Lecture Notes in Mathematics, vol. 303, pp. 111-124, 1972.
[17] R. Jayakumar, K. Thulasiraman, and M.N.S. Swamy, “On Maximal Planarization of Nonplanar Graphs," IEEE Trans. Circuits and Systems, vol. CAS-33, no. 8, 843-854, 1986.

[18] R. Jayakumar, K. Thulasiraman, and M.N.S Swamy, “O(n^2) Algorithms for Graph Planarization," Technical Report CSD-88-01, Dept. Computer Science, Concordia Univ., 1988.
[19] N. Chiba, I. Nishioka, and I. Shirakawa, “An Algorithm of Maximal Planarization of Graphs," Proc. IEEE Int. Symp. on Circuits and Systems, pp. 649 652, 1979.
[20] M. Marek-Sadowska, “Planarization Algorithms for Integrated Circuits Engineering," Proc. IEEE Int. Symp. on Circuits and Systems, pp. 919-923, 1978.
[21] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa, “A Linear Algorithm for Embedding Planar Graphs Using PQ-Trees," J. of Computer and System Sciences, vol. 30, no. 1, pp. 54-76, 1985.
[22] K. Booth and G. Lueker, “Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms," J. of Computer and System Sciences, vol. 13, pp. 335-379, 1976.
[23] F. István, “On straight-line representation of planar graphs”, Acta Sci. Math. (Szeged), 11: 229–233
[24] B. Becker and G. Hotz, “On The Optimal Layout of Planar Graphs with Fixed Boundary," SIAM J. Computing, vol. 16, no. 5, pp. 946-972, 1987.
[25] P. Eades and N. Wormald, “Fixed Edge Length Graph Drawing is NP- hard," Discrete Applied Mathematics, vol. 28, pp. 111-134, 1990.
[26] J. Alam, F. Brandenburg, and S. Kobourov, “Straight-Line Grid Drawings of 3-Connected 1-Planar Graphs,” Graph Drawing. GD 2013. Lecture Notes in Computer Science, vol 8242. Springer, Cham
[27] H. de Fraysseix, J. Pach, R. Pollack, “How to draw a planar graph on a grid,” Combinatorica 10(1), 41–51 1990.
[28] W. Schnyder, “Embedding planar graphs on the grid,” Symposium on Discrete Algorithms. pp. 138–148 1990.
[29] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, 1979.
[30] S. Hong, P. Eades, G. Liotta, S. Poon, “Fary's theorem for 1-planar graphs,” Lecture Notes in Computer Science (LNCS), 7434, 335-346. 2012.
[31] S. Bhowmick and S. Shontz, “Towards High Quality, Untangled Meshes via a Force-Directed Graph Embedding Approach,” Procedia Computer Science 2008.
[32] L. Freitag, P. Plassmann, “Local optimization-based untangling algorithms for quadrilateral meshes,” Proc. of the Tenth International Meshing Roundtable, Sandia National Laboratories, 2001
[33] J. Escobar, E. Rodriguez, R. Montenegro, G. Montero, J. Gonzalez-Yuste, “Simultaneous untangling and smoothing of tetrahedral meshes,” Comput. Method. Appl. M. 192 (2003) 2775–2787.
[34] H. Djidjev, “Force-directed methods for smoothing unstructured triangular and tetrahedral meshes,” Proc. of the Ninth International Meshing Roundtable, Sandia National Laboratories, 2000

APPENDIX

Appendix A: Source Code

```
import java.lang.Math;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.util.Scanner;
import java.lang.String;
import java.io.File;

public class Fruchterman-Reingold_algorithm_from_vtk_final
{
public static void main(String [] args)
{
      Scanner scan = new Scanner(System.in);
      System.out.print("Please enter the filepath to the input
file: ");
      String str = scan.next();
      System.out.print("Please enter the number of dimensions
specified in the file: ");
      int dimension = scan.nextInt();

      File f = new File(str);
      Scanner scan2;
      try
      {
            scan2 = new Scanner(f);
      }
      catch(Exception ex)
      {
            return;
      }
      String dummy = scan2.nextLine();
      dummy = scan2.nextLine();
      dummy = scan2.nextLine();
      dummy = scan2.nextLine();

      dummy = scan2.next();
      int numv = scan2.nextInt();
      dummy = scan2.next();

      double[][] verts = new double[numv][dimension];
      int[] degree = new int[numv];
      int[][] adj = new int[numv][numv];

      for(int i = 0; i < numv; i++)
      {
            for(int j = 0; j < dimension; j++)
            {
                  verts[i][j] = scan2.nextDouble();
            }
      }

      dummy = scan2.next();
```

```
int numc = scan2.nextInt();
int nume2 = scan2.nextInt();
int nume = 0;
int numvc = nume2/numc;

for(int i= 0; i < numc; i++)
{
      int[] vs = new int[numvc];
      for(int j = 0; j < numvc; j++)
      {
            vs[j] = scan2.nextInt();
      }
      for(int j = 1; j < numvc - 1; j++)
      {
            if(adj[vs[j]][vs[j+1]] != 1)
            {
                  adj[vs[j]][vs[j+1]] = 1;
                  adj[vs[j+1]][vs[j]] = 1;
                  nume++;
            }
      }
      if(adj[vs[numvc-1]][vs[1]] != 1)
      {
            adj[vs[numvc-1]][vs[1]] = 1;
            adj[vs[1]][vs[numvc-1]] = 1;
            nume++;
      }
}

int[][] edges = new int[nume][2];
int ed = 0;

//adj mat
try
{
      BufferedWriter bw = new BufferedWriter(new
FileWriter("adjmat.dat"));

      for(int i = 0; i < numv; i++)
      {
            for(int j = 0; j < numv; j++)
            {
                  bw.write(adj[i][j] + " ");
                  if(j > i)
                        continue;
                  if(adj[i][j] == 1)
                  {
                        edges[ed][0] = i;
                        edges[ed][1] = j;
                        ed++;
                  }
            }
            bw.newLine();
      }
      bw.close();
}
```

```
      catch(Exception e1)
      {
            System.out.println("Error in writing adjmat file");
            return;
      }

      dummy = scan2.next();
      int num = scan2.nextInt();

      for(int i = 0; i < num + 4; i++)
      {
            dummy = scan2.nextLine();
      }

      int[] frameverts = new int[numv];
      int[] adjverts = new int[numv];
      int[] procverts = new int[numv];

      for(int i = 0; i < numv; i++)
      {
            num = scan2.nextInt();
            if (num == 1)
            {
                  frameverts[i] = 1;
            }
            else
            {
                  frameverts[i] = 0;
            }
      }
      for (int e = 0; e < nume; e++)
      {
            degree[edges[e][0]]++;
            degree[edges[e][1]]++;
            if(frameverts[edges[e][0]] == 1 &&
frameverts[edges[e][1]] != 1)
            {
                  adjverts[edges[e][1]] = 1;
            }
            else if(frameverts[edges[e][1]] == 1 &&
frameverts[edges[e][0]] != 1)
            {
                  adjverts[edges[e][0]] = 1;
            }
      }


      System.out.print("Please enter the number of iterations:
");
      int iterations = scan.nextInt();

      double fxmin, fxmax, fymin, fymax; //frame boundaries
      System.out.print("Please enter the xmin value for the
frame: ");
      fxmin = scan.nextDouble();
```

```
      System.out.print("Please enter the xmax value for the
frame: ");
      fxmax = scan.nextDouble();
      System.out.print("Please enter the ymin value for the
frame: ");
      fymin = scan.nextDouble();
      System.out.print("Please enter the ymax value for the
frame: ");
      fymax = scan.nextDouble();

      System.out.println("Please enter the correct number for
the type of mesh you are processing:");
      System.out.println("0: Corners-attract style mesh");
      System.out.println("1: Fixed-frame style mesh");
      System.out.println("2: General case mesh");
      System.out.print("Please enter your selection: ");


      int choice = scan.nextInt();

      double width = fxmax-fxmin;
      double height = fymax-fymin;
      double area = width * height;
      double k = Math.sqrt(area/numv);
      double[][] vpos = new double[numv][2];

      for(int i = 0; i < numv; i++)
      {
            vpos[i][0] = verts[i][0];
            vpos[i][1] = verts[i][1];
      }

      double[][] vdisp = new double[numv][2];

      for (int i = 0; i < iterations; i++)  //Main Fruchterman-
Reingold code
      {
            for (int v = 0; v<numv; v++)     //repulsive forces
            {
                  vdisp[v][0] = 0;
                  vdisp[v][1] = 0;
                  for (int u = 0; u<numv; u++)     //loop
through every vertex
                  {
                        if(v == u) //vertex cannot repel itself
                              continue;
                        double[] delta = new double[2];
                        delta[0] = vpos[v][0] - vpos[u][0];
                        delta[1] = vpos[v][1] - vpos[u][1];

                        if(delta[0] == 0)     //act as if they
are very slightly apart if they are on the same spot
                              delta[0] = .001;
                        if(delta[1] == 0)
                              delta[1] = .001;
```

```
                        double magd = (Math.pow(delta[0],2) +
Math.pow(delta[1],2));      //distance between
                        //magd *= 20;    //reduce intensity for
this

                        vdisp[v][0] +=
(Math.pow(k,2)/magd)*(delta[0]/magd);
                        vdisp[v][1] +=
(Math.pow(k,2)/magd)*(delta[1]/magd);
                  }
            }

            for(int e = 0; e<nume;e++)  //attractive forces
            {
                  int v = edges[e][0];
                  int u = edges[e][1];

                  double[] delta = new double[2];
                  delta[0] = vpos[v][0] - vpos[u][0];
                  delta[1] = vpos[v][1] - vpos[u][1];

                  if(delta[0] == 0)     //act as if they are
very slightly apart if they are on the same spot
                        delta[0] = .001;
                  if(delta[1] == 0)
                        delta[1] = .001;

                  double magd = (Math.pow(delta[0],2) +
Math.pow(delta[1],2));

                  double mult1 = 1;
                  double mult2 = 1;
                  if(frameverts[u] != 1)
                        mult1 = 1;
                  if(frameverts[v] != 1)
                        mult2 = 1;

                  vdisp[v][0] -=
(Math.pow(magd,2)/k)*(delta[0]/magd)/mult1;
                  vdisp[v][1] -=
(Math.pow(magd,2)/k)*(delta[1]/magd)/mult2;

                  vdisp[u][0] +=
(Math.pow(magd,2)/k)*(delta[0]/magd);
                  vdisp[u][1] +=
(Math.pow(magd,2)/k)*(delta[1]/magd);


            }

            //move vertices
            for (int v = 0; v<numv; v++)
            {
                  //int adjverts =
                  if(frameverts[v] == 1 || adjverts[v] != 1)
                  {
```

```
                        continue;
                  }

                  double magv =
Math.sqrt(Math.pow(vdisp[v][0],2) + Math.pow(vdisp[v][1],2));

                  if(procverts[v] == 1 && choice == 1)   //reduce
movement for vertices already processed.
                  {
                        magv *= 10;
                  }

                  vpos[v][0] += vdisp[v][0]/magv;
                  vpos[v][1] += vdisp[v][1]/magv;

            }
            if(i % 1000 == 0)
            {
                  for(int counter2 = 0; counter2 < numv;
counter2++)
                  {
                        //continue;
                        if(adjverts[counter2] == 1)
                        {
                              //adjverts[counter2] = 0;
                              procverts[counter2] = 1;
                        }
                  }
                  for(int counter = 0; counter < nume;
counter++)
                  {
                        int v = edges[counter][0];
                        int u = edges[counter][1];
                        if((frameverts[v] == 1 || procverts[v] ==
1 || adjverts[v] == 1))
                        {
                              adjverts[u] = 1;
                        }
                        else if((frameverts[u] == 1 ||
procverts[u] == 1 || adjverts[u] == 1))
                        {
                              adjverts[edges[counter][0]] = 1;
                        }
                  }
            }
      }


      try
      {
      BufferedWriter bw3 = new BufferedWriter(new
FileWriter("verts.dat"));
      for(int v = 0; v<numv; v++)
      {
            bw3.write(vpos[v][0] + " " + vpos[v][1] + " " +
"0");
```

```
            bw3.newLine();
        }
        //bw.close();
        bw3.close();
        }
        catch(Exception ex)
        {
            return;
        }
    }
}
```