# Geometry Processing Project Report
**Darryl Hill**

**Abstract**

Delaunay meshes are superior to non-Delaunay meshes for certain applications. For example, Delaunay meshes increase the accuracy of the heat method for computing geodesic distances. As well the cotangent Laplace-Beltrami operator has non-negative weights on a Delaunay mesh. Also, some popular parameterization techniques, such as discrete harmonic mapping, produce more stable results on Delaunay meshes.

What is more, we can construct a Delaunay mesh from an input mesh while preserving the geometry of the input mesh. The outputed DM has only $O(Kn)$ vertices, where $n$ is the number of vertices in the mesh and $K$ is a constant that depends on mesh input parameters. $K$ is proportional to the ratio of max to min edge lengths in the mesh, and inversely proportional to the minimum angle in the input mesh. In theory high quality meshes produce an output mesh with less vertices, while in practice one bad triangle can result a large inflation in the number of additional points generated.

.

## 1 Objective

This project is based on the paper "Efficient Construction and Simplification of Delaunay Meshes" [6]. As the title implies, the authors design an algorithm for converting an input mesh into a Delaunay mesh, while maintaining the same geometry as the input mesh.

The authors implement "Algorithm 2" from the paper [6], which takes an input mesh and generates a Delaunay mesh. Their goals were

1. Given arbitrary mesh, generate Delaunay mesh.
2. Preserve mesh geometry.
3. Output mesh is reasonable size.
4. Algorithm terminates.
5. Test different data structures (queue, stack) and measure performance.

The authors accomplished from 1 to 4, and determined that the stack performed better than the queue as a data structure for processing non-Delaunay edges (NDE's).

**My Objectives** My objectives were to improve the performance and output quality of the algorithm. I contacted the authors, but unfortunately they could not publish the source code to their algorithm at this time. So I would implement the algorithm before attempting an improvement. As such my goals were as follows:

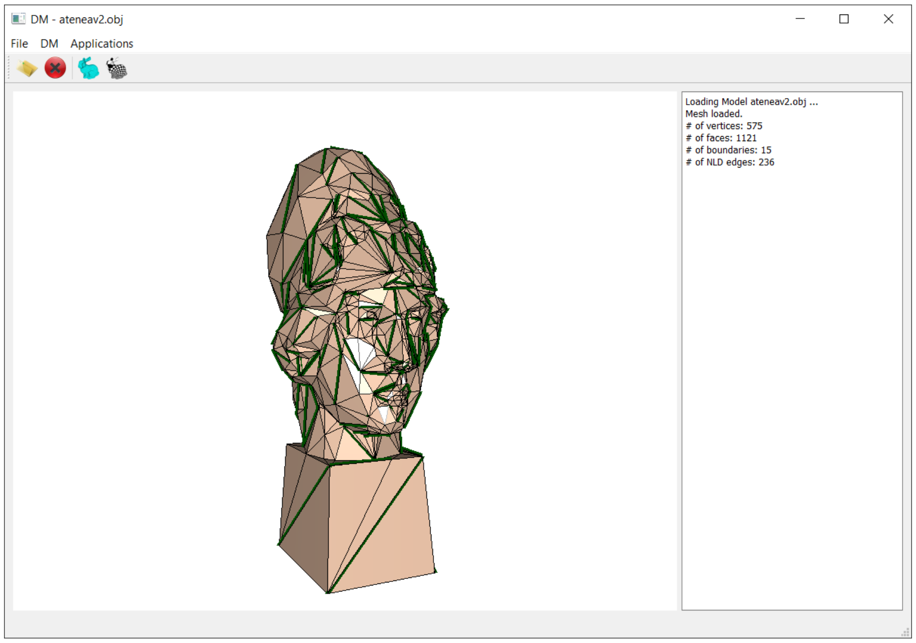
1. Implement "Algorithm 2" as written.
2. Implement and test a max priority queue based on edge length for edge processing.
3. Investigate what, if any, heuristics could speed up the processing.
4. Investigate the theory behind the Delaunay mesh construction to try and find additional ways to speed up or simplify the algorithm.

I was successful in 1-3, but failed to find any meaningful results for 4.

## 2 Methodology

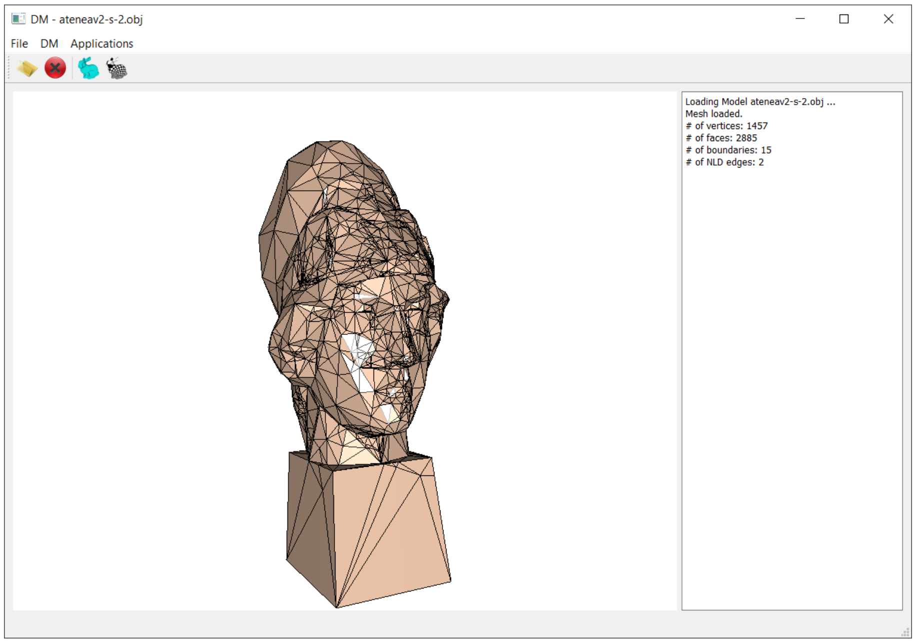
### 2.1 Identifying NDE's

To identify a non-Delaunay edge, the program examined the two angles opposite the edge. If the sum of the two angles was $< \pi$, then it was considered a non-



```
Loading Model ateneav2.obj ...
Mesh loaded.
# of vertices: 575
# of faces: 1121
# of boundaries: 15
# of NLD edges: 236
```

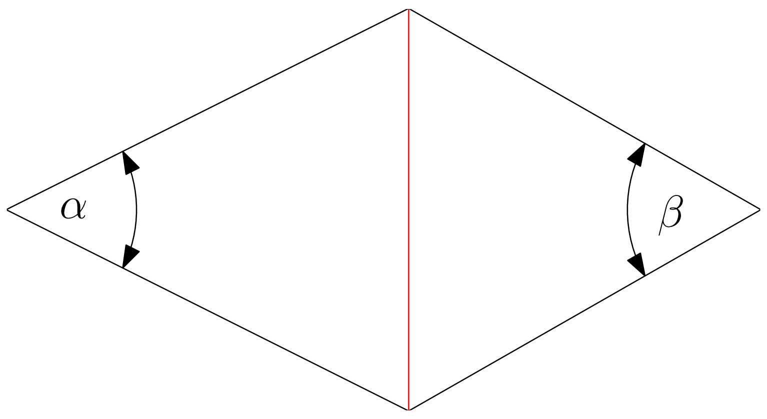
(a) Green edges are non-Delaunay edges.



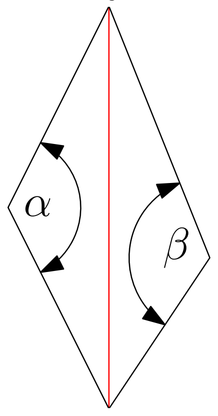
```
Loading Model ateneav2-s-2.obj ...
Mesh loaded.
# of vertices: 1457
# of faces: 2885
# of boundaries: 15
# of NLD edges: 2
```

(b) Geometry is identical.

*Figure 1: Algorithm 2 turns (1a) into (1b).*



(a) If $\alpha + \beta \leq \pi$ the red edge is a Delaunay edge.



(b) If $\alpha + \beta > \pi$ the red edge is a non-Delaunay edge.

*Figure 2*

Delaunay edge. See Fig. 2.

## 2.2 Making Samples

The Delaunay mesh is the dual of the geodesic Voronoi diagram (GVD), which is a Voronoi diagram on a 2-manifold. However, the Delaunay triangulation for a GVD does not always exist. It requires that the generating points of the GVD are sufficiently dense that the geodesic paths between them are unique. This is called the closed ball property[1].

[1]The closed ball property is actually defined differently, but it is implied by the above condition.

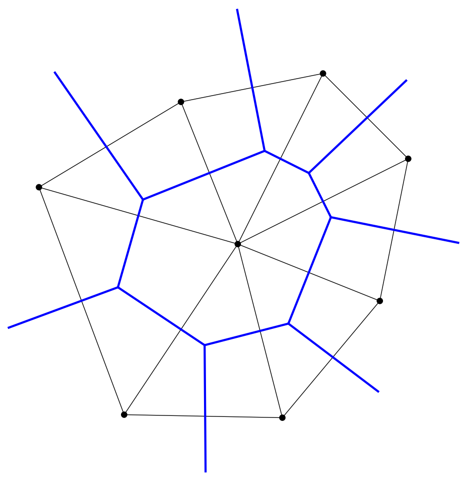


*Figure 3: Voronoi cell within the one-ring neighbourhood of its generating point.*

One way to achieve the closed ball property on a mesh is to ensure that the Voronoi cell of a point is within its one-ring. See Fig. 3.

Based on the minimum edge length and minimum angle in the mesh, the authors calculate two constants. In order to maintain the original geometry, new points are added along edge. These constants are used to determine the density of points along all the edges of the mesh needed to ensure each Voronoi cell is within the one-ring of its generating point.

However, this is a stronger property than the closed ball property. As a result, it is not necessary to add all the sample points to the mesh to achieve a Delaunay mesh. Thus the algorithm finds all NDE's, stores them in a data structure, and splits them one at a time by adding a sample point somewhere along the edge are re-triangulating. The algorithm then checks the two resulting edges and the surrounding edges for the Delaunay property. Any edges that became non-Delaunay by the addition of the vertex are added to the processing data structure to be split or flipped at a later time.

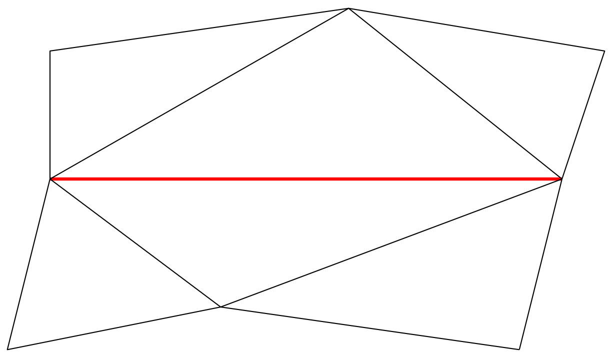


*Figure 4: Unfolding the neighbourhood of an edge (in red) into the plane.*

## 2.3 Choosing Samples

Adding a point could split a NDE into one or two Delaunay edges, but it could induce other edges to become NDE's. If all the samples in the mesh were added, then we provably have a Delaunay mesh, but the goal was to create a Delaunay mesh by adding as few vertices as possible.

To that end, when adding a sample point, the authors would choose a point that creates the fewest NDE's in the mesh. To test the samples on a particular edge $e$, they take the two faces adjacent to $e$, as well as the 4 additional faces neighbouring those two faces, and unfold them into the plane. See Fig. 4.

To implement the unfolding we simply measure the lengths and angles and reconstruct an image of the six triangles in 2 dimensions. The sample points are likewise transformed, then tested for how many circumcircles they are in.

From there each sample point earns a score based on whether they fall into the circumcircle of the various faces. If a point is not within the circumcircle of a triangle, then adding that point would not cause the shared edge of the triangle to become non-Delaunay. Fig. 5

## 2.4 Flippable Edges

When a sample point is added to the mesh, and the mesh retriangulated, the original edge is split into two edges. In addition, there are two new edges that

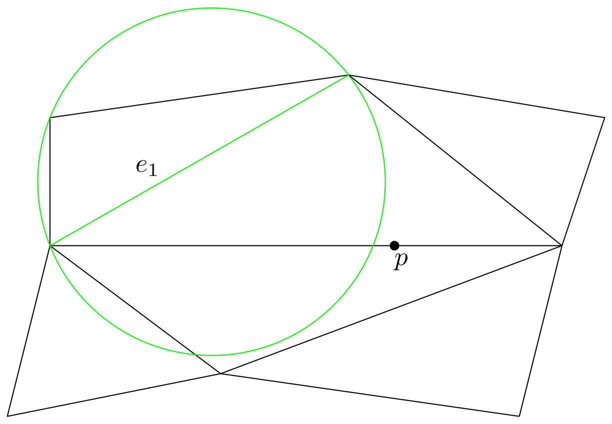


*Figure 5: Adding point $p$ leaves edge $e_1$ a Delaunay edge.*

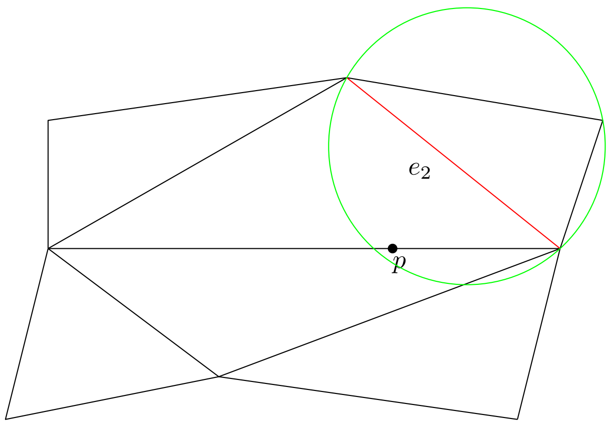


*Figure 6: Adding point $p$ makes $e_2$ a non-Delaunay edge.*

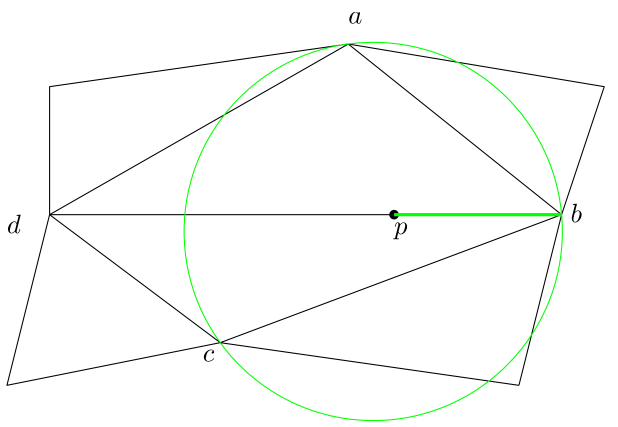


*Figure 7: Adding $p$ within the circle through $a, b$ and $c$ means edge $(p, b)$ is Delaunay.*

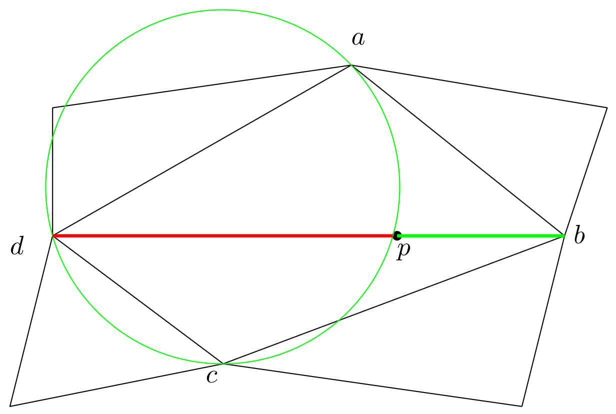


*Figure 8: Point $p$ is not in the circle through $a, c$ and $d$ and thus $(p, d)$ is non-Delaunay.*

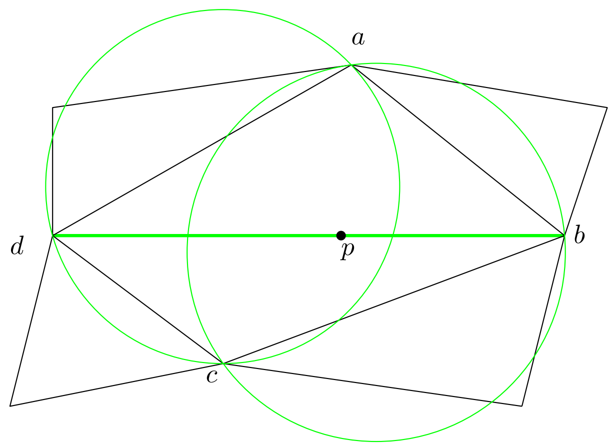


*Figure 9: If $p$ is within circles through $a, b$ and $c$ and through $a, c,$ and $d$, then both $(d, p)$ and $(p, b)$ are Delaunay edges.*

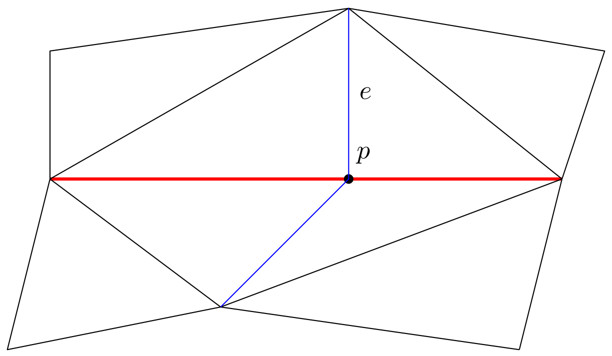


*Figure 10: When adding p, the blue edges are new, flippable edges.*

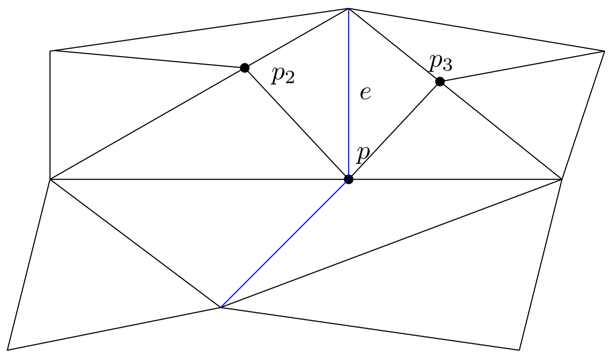


*Figure 11: If we add $p_2$ and $p_3$, now edge e can become NDE.*

are added. See Fig. 10.

Since these edges lie across an existing face, they have a diametric angle of $\pi$. When they are first added they are always Delaunay edges. However, as more points are added they can become non-Delaunay. However, instead of splitting these edges, we can fip them to make them Delaunay, since it will not change the geometry of the mesh. See Figs. 10, 11, and 12.

## 2.5 The Algorithm

I implemented Algorithm 2 from the aforementioned paper, which I will briefly describe.

1. Calculate all sample points for all edges in the mesh

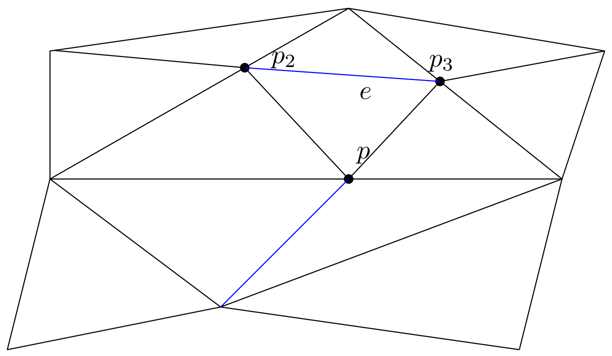


*Figure 12: Since the diametric angle of $e$ is $\pi$, we can flip $e$ while maintaining the geometry.*

2. Find all NDE's in the mesh and put them in a data structure $DS$.
3. while $DS$ is not empty:
   (a) Process next NDE in $DS$ by adding a sample point to it, then retriangulating.
   (b) Examine edges in faces adjacent to the split edge.
   (c) If it is a flippable NDE, then flip it.
   (d) Add non-flippable non-Delaunay edges to $DS$.

As written the algorithm did not quite work. After adding a vertex, the authors check in the immediate neighbourhood for any NDE's that were created. However, an added vertex can induce an NDE that is not in the immediate neighbourhood. So the algorithm required multiple iterations to complete (though it is asymptotically the same). Ultimately the omission was somewhat minor.

The output was written to a .OBJ file. I compiled the data in these files to obtain statistics.

**Changes / Improvements**

1. New data structure (priority queue) implemented and tested.
2. Implemented heuristics in an attempt to speed up processing.

1. **Data Structure**: While processing non-Delaunay edges (NDEs), the algorithm sometimes creates more NDEs in the surrounding neighbourhood (though it still provably terminates). The authors remark that the algorithm adds less vertices using a stack to process newly created edges, rather than a queue. However, they give no theoretical justification for this phenomenon, instead relying on empirical evidence.

I had believed that the phenomenon they witnessed is a function of processing the edges in order of length, from longest to shortest. Splitting a NDE sometimes creates another NDE, which is often shorter. Splitting the long edge first created fewer large angles, which would, in the long run, make less NDE's. See Fig. 13. Processing the edges on a stack provides an inadvertent largest to smallest ordering for these local cases, the majority of the time. I processed the edges in order of length by implementing a priority queue. This will add a $\log j$ factor of computational overhead, where $j$ is the number of NDE's processed.

Ultimately the priority queue performed about on par with the stack when all the heuristics were applied. The stack was consistently better than the priority queue when the heuristics were less than optimal. The priority queue out-performed the queue in most cases.

2. **Heuristics:** As well I made other adjustments to vertex placement when dealing with adjacent NDE's. I provided an additional weighting heuristic when deciding what sample point to add to the mesh, as well as a heuristic for selecting the median sample among equally weighted vertices.

## 3 Implementation

The implementation is in C++. For mesh handling and data storage I used OpenMesh, based on its superior documentation. For the geometric computations I used a combination of a library called "triangle properties" [4] and a class I wrote myself called Geom2D.

Some of the datasets were pulled from the website TF3DM [1]. These include the elephant and the boar.

For test purposes I used the bunny.obj file that was generated from assignment 1. I chose a sufficiently smooth version and used that, although you may still notice some minor imperfections.

I also used a statue of the head of Athena called "ateneav.obj", that I got from a link on cuLearn to Robin Bing-Yu Chen's webpage [2]. I chose the low poly version to make it easier to distinguish individual edges. I also loaded it into OpenMesh and wrote it out to a file called "ateneav2.obj". This eliminated the normals that were included in the original mesh. Having a raw mesh to work with made it easier to pick out the difference between the input and output mesh.

I also used a very simple cube model that I created myself to help with the fine grained testing of the algorithm. It helped verify that the program was splitting the edges it was supposed to. I have not included it in the tables, since there were not enough faces to obtain interesting data, although I did include the file in the datasets.

## 4 Results

I compiled the results of using the different heuristics, as well as the data structures, into 3 tables (for the 3 data sets used). They are tables 1, 2, and 3. In general the heuristics worked quite well, while the priority queue had mixed success.

### 4.1 Large Data Sets

This particular implementation is not always suitable for large data sets. The program initially generates constants that determine the number of sample points added to an edge. These constants are inversely proportional to the minimum angle and minimum edge length. So on a mesh with a very small angle, the number of samples generated can average 100 000 per edge. That is at a minimum of 6 bytes per point (3 doubles to store location). If there 5000 vertices there can be 15 000 edges, which can require as many as 9 000 000 000 bytes, or roughly 9 gigabytes.

The largest data set I used was the bunny.obj file, which was 7473 edges, but the sample size averaged 1 or 2 thousand. I attempted ateneam2.obj, which had approximately 22 000 edges, with sample sizes in some cases over 200 000. This was killed off by the system before completion as it consumed too much memory.

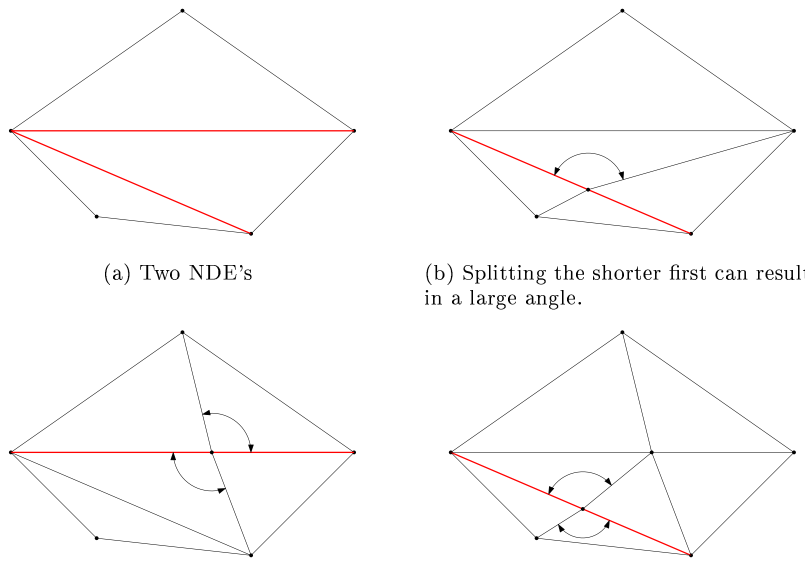
There are a couple of approaches that could mitigate this. We could generate samples only when we process an edge. It would add some complexity to the algorithm, but it would functionally be the same. Another solution is to store only the first and last sample point, and generate the other points as they are needed. They are equally spaced, with the exception of the first and last point, so it would not be too difficult. Unfortunately, factoring in time for debugging, at this late time I will not be able to implement either of these solutions.

## 4.2 Priority Queue

In two of the three data sets used for testing, the priority queue edged out the stack in terms of performance when the two heuristics were being used. Interestingly, the stack outperformed the priority queue when only one or none of the heuristics were applied. That seems to imply that the stack performs consistently better than the priority queue over a wider number of situations.

Tests on larger data sets may put this issue to rest, for the time being though I would rank stack over the priority queue.

**Observations** Splitting the larger edges first tends to minimize the maximum angle produced. See Fig. 13



(a) Two NDE's

(b) Splitting the shorter first can result in a large angle.

(c) Splitting the longer first tends to minimize the maximum angle produced.

(d) Compared to 13b the max angle is less.

*Figure 13: Reasoning behind using the priority queue*

## 4.3 Heuristics

### 4.3.1 Weighted Scoring

In the author's paper, when processing a NDE, they select an appropriate sample point for splitting by scoring each point according to how many NDE's will be created by adding that point. The lowest score is chosen and that point is added to the mesh. The authors propose that each NDE that is created is weighted equally. We included this heuristic, as well as two others.

The two additional weighting systems both involving an increased weight given to ensuring the edge we are splitting is split into two Delaunay edges. Adding a sample point splits the current NDE in two edges. For each of these two edges, if they were Delaunay we scored 1.5. The other 4 edges in the current neighbourhood scored 1 if they remained Delaunay as a result of the added sample point.

Additionally, for the two segments of the split edge, we tried a scoring system that gave 5 points for if both those edges were Delaunay. Since the other 4 edges could score at most 4, this ensured that the sample points that split the original edge into two Delaunay edges were definitely chosen over any other sample point.

We rank them in terms of the total number of edges in the mesh after using them, where the lowest number results in the highest rank.

1. Always center
2. Weighted center
3. Equal weighting

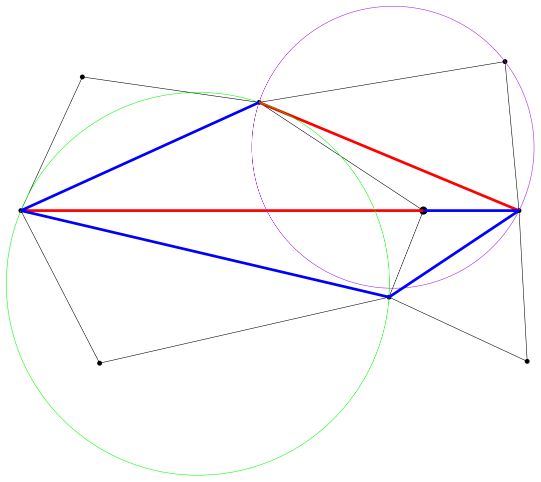
**Observations** Failing to split the current edge into two Delaunay edges can result in repeated splitting of the same edge. In essence, we do not fix the edge that we were trying to fix. We merely shorten it. See Fig. 14.

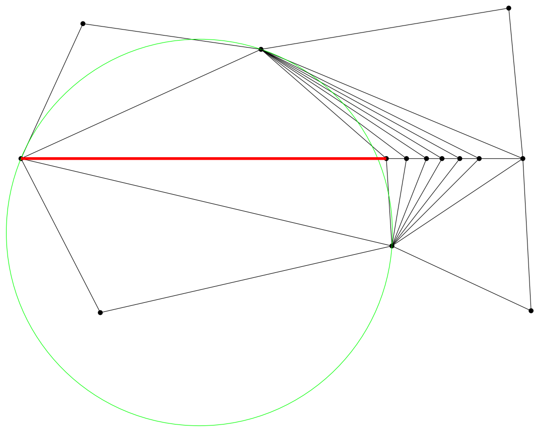
### 4.3.2 Median

Of all the sample points that scored the highest, we choose the median of these to add to the mesh. This is a simplistic heuristic that perhaps the authors use.[2] We found that it performed quite well, better than the Always Center weighting heuristic. Of course, both of these heuristics together were better than either alone.

**Observations** Splitting an edge results in new edges with new angles. Splitting at the median point results in an angle that approaches the minimum maximal angle. Theoretically this heuristic could be improved by choosing the sample point that actually minimizes that maximal angle. Choosing the median is a quick and dirty way to approximate this. See Fig. 15.

[2]In the paper they only mention selecting one of the highest ranking points (ranked by minimal NDE's generated).


(a)


(b)

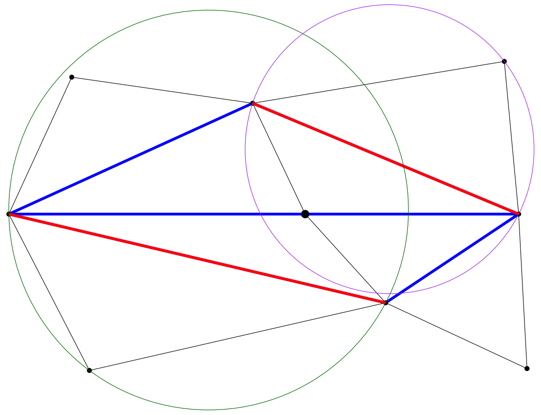
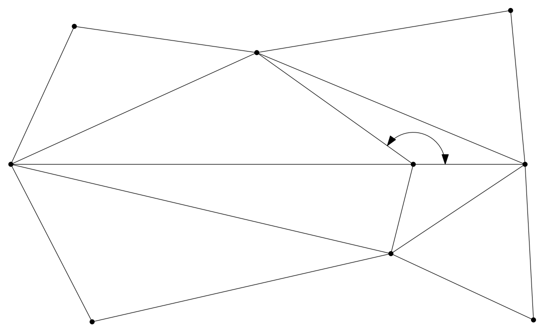

(c)

*Figure 14: Red are NDE's, blue are Delaunay edges. Failing to split the edge into Delaunay edges (15a) can result in multiple added points (15b). The solution (14c) is to always split it.*


(a)

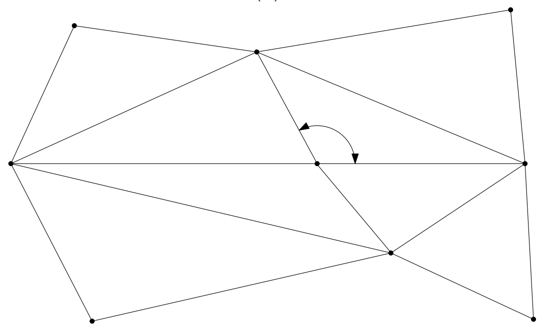

(b)

*Figure 15: Choosing the median point minimizes the maximum angle.*

#### 4.3.3 Comparision With Previous Results

The authors make no mention of heuristics in their paper, but that does not necessarily mean they did not use any. Unfortunately the points sets that they used in their paper to compile data from are far too large to run on this implementation. Additionally, although the authors have an executable file of their algorithm on their website, attempting to run the Delaunay mesh algorithm results in it crashing. Perhaps it was compiled on an incompatible system. Additionally, when contacted, they told me they could not publish their source code at this time.

As a result, there is no way currently to compare our results to those of the authors.

### 4.4 Theoretical Results

Unfortunately the implementation took longer than I expected, thus limiting the time spent exploring the theory behind Delaunay meshes and this technique in particular. So there are no theoretical results per say, besides the observations leading to the heuristics.

## 5 Documentation

Add a "Documentation" section, describing the main components of the code and how to compile/run it.

### 5.1 Compiling

I compiled the program in Linux using QT. It must be linked to the library OpenMesh; all other files are included.

I have included all the files QT generated. Many are likely redundant, but if you happen to load it into QT it should be able to make sense of it.

I have included the executable for the program - maybe it will run. I do not have a lot of experience with ensuring C++ code will compile and run

*Table 1: Elephant Model - 1148 faces*
*min angle: 0.0658129*
*max length: 265.138*
*min length 1.46895*

| Data Structure | Weight | Median Point | Faces |
|---|---|---|---|
| stack | always center | Yes | 3360 |
| stack | all equal | Yes | 3638 |
| priority queue | always center | Yes | 3480 |
| priority queue | weighted center | Yes | 3488 |
| queue | always center | Yes | 3748 |
| priority queue | all equal | Yes | 3758 |
| queue | weighted center | Yes | 3778 |
| queue | all equal | Yes | 3816 |
| stack | weighted center | Yes | 4010 |
| stack | always center | No | 4206 |
| stack | weighted center | No | 4212 |
| queue | always center | No | 4368 |
| queue | weighted center | No | 4420 |
| priority queue | always center | No | 4656 |
| priority queue | weighted center | No | 4970 |
| stack | all equal | No | 19256 |
| queue | all equal | No | 39988 |
| priority queue | all equal | No | 41426 |

*Table 2: Athena - 1121 faces*
*min angle: 0.0524457*
*max length: 2514.2*
*min length 35.3139*

| Data Structure | Weight | Median Point | Faces |
|---|---|---|---|
| priority queue | always center | Yes | 2729 |
| priority queue | all equal | Yes | 2763 |
| queue | always center | Yes | 2855 |
| queue | weighted center | Yes | 2855 |
| stack | always center | Yes | 2885 |
| priority queue | weighted center | Yes | 2957 |
| stack | weighted center | Yes | 2987 |
| priority queue | always center | No | 3001 |
| priority queue | weighted center | No | 3005 |
| stack | all equal | Yes | 3035 |
| queue | all equal | Yes | 3055 |
| stack | always center | No | 3067 |
| stack | weighted center | No | 3091 |
| queue | weighted center | No | 3161 |
| queue | always center | No | 3193 |
| queue | all equal | No | 17575 |
| stack | all equal | No | 17951 |
| priority queue | all equal | No | 14223 |

*Table 3: Bunny Model - 4968 faces*
*min angle: 0.0383127*
*max length: 0.0215783*
*min length 0.000758634*

| Data Structure | Weight | Median Point | Faces |
|---|---|---|---|
| stack | always center | Yes | 7030 |
| stack | weighted center | Yes | 7030 |
| priority queue | always center | Yes | 7044 |
| priority queue | weighted center | Yes | 7044 |
| priority queue | all equal | Yes | 7078 |
| queue | always center | Yes | 7092 |
| queue | weighted center | Yes | 7092 |
| stack | all equal | Yes | 7184 |
| queue | all equal | Yes | 7192 |
| stack | always center | No | 7290 |
| stack | weighted center | No | 7290 |
| queue | always center | No | 7342 |
| queue | weighted center | No | 7362 |
| priority queue | always center | No | 7342 |
| priority queue | weighted center | No | 7350 |
| stack | all equal | No | 14146 |
| queue | all equal | No | 15352 |
| priority queue | all equal | No | 16030 |

cross-platform. However, here is what I know.

Navigate to qt-3/DelMesh. There is an executable you can try. There is also a makefile you can try, but you will likely have to delete it and generate another one.

To generate a makefile is: qmake DelMesh.pro

That will complete almost instantly, at which point you can try typing "make". With any luck, and if OpenMesh is installed, this should generate a new executable that will run on your system.

## 5.2 Running DelMesh

It is a standard executable, so ./DelMesh will run it. However, you must specify at least an input file. All other arguments are optional. If you do not provide an output file, it will rewrite the input file.

- -i <input file>
- -o <output file>
- -ds <"queue"|"stack"|"pqueue">
- -s <"0"|"1"|"2">
- -nomedian
- -all

-ds is the data structure you would like to use.

-s is the scoring type. 0 is equally weighted (as in the original paper). 1 is for a 1.5 weighting to a Delaunay split edge. 2 means we always split the edge into 2 Delaunay edges whenever possible.

-nomedian turns off the median heuristic (which is on by default).

-all will systematically generate all combinations. It might take a little while. When using -all your output file will automatically be appended with the proper suffix to indicate which choices were made. For example

```
elephant-s-0-m.obj
```

indicates the mesh was processed using a stack, with scoring type 0, and the median heuristic activated. elephant-s-0.obj is the same except with no median heuristic.

Note that -all will nullify any other inputs (except for the input and output files).

Example inputs:

```
./DelMesh -i elephant2.obj -o output/elephant.obj -s 2 -ds pqueue
```

will load elephant2.obj, process it using a priority queue, with the score type of 2, and save it to output/elephant.obj.

```
./DelMesh -i elephant2.obj -o output/elephant.obj -all
```

will run every combination of input on elephant2.obj and output all the files to the output directory.

## 6 Conclusion

The algorithm performed admirably and correctly. However, some implementation details could be improved, particularly if we wish to work on large data sets.

When all the heuristics were applied, the priority queue and stack were roughly neck and neck. When the other heuristics were minimized, the stack performed better most of the time. So the stack was consistently better.

The two observed heuristics both performed well. What, if any, heuristics were implemented in the author's source code is unknown. It would have been nice to find the data sets they used and compare this algorithm to their reported numbers. Unfortunately time became an issue. It could be they implemented the median heuristic, or something similar, but did not mention it. However, their description specifically mentioned choosing a sample point that minimized the number of created NDE's, which is not what our center weighting heuristic does. In our heuristic we may create more NDE's short term, but avoid creating more NDE's long term.

I did put some time in to trying to find theoretical results, but other than a few observations came up with no concrete improvements in the analysis.

## References

[1] 3d-models. `http://tf3dm.com/3d-models/lowpoly`. Accessed: 2016-03-15.

[2] Robin Bing-Yu Chen. Computer gaphics.

[3] Jean-Daniel Boissonnat and Steve Oudot. Provably good sampling and meshing of lipschitz surfaces. In *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, SCG '06, pages 337–346, New York, NY, USA, 2006. ACM.

[4] John Burkardt. Triangle properties.

[5] Yong-Jin Liu, Zhan-Qing Chen, and Kai Tang. Construction of iso-contours, bisectors and voronoi diagrams on triangulated surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(8):1502–1517, 2011.

[6] Yong-Jin Liu, Chun-Xu Xu, Dian Fan, and Ying He. Efficient construction and simplification of delaunay meshes. *ACM Trans. Graph.*, 34(6):174:1–174:13, October 2015.