# AE623
# Project 1: Mesh Generation

Ben Bates, Thomas Greenwald, Vishwa Mohan Tiwari,
Moon Bakaya Hazarika, and Elizabeth Wraback
*EcoLine*

February 1, 2024

**Mesh generation is necessary for almost all CFD numerical schemes. For a given three-piece airfoil, we generated a coarse, unstructured, triangular mesh with 1554 elements on which processing was conducted. We performed uniform refinement at increasing levels of resolution (8k, 32k, and 128k elements), where each mesh element was split into four elements through edge bisection. We also implemented local refinement of the mesh, which was then smoothed using a weighted average of nearby node coordinates. The meshes are verified with the error being zero on each element.**

## 1 Introduction

Meshes are necessary to discretize the computational domain for implementation of various numerical schemes. Therefore, mesh generation is an important first step of solving any CFD problem.

A mesh is a subdivision of the computational domain into smaller, usually regularly-shaped areas (in 2D) or volumes (in 3D). In 2D, these areas are referred to as elements. These meshes can be structured or unstructured, with the former having node connectivity implied and the latter having connectivity explicitly stored [1].

Unstructured meshes can be created using various methods. Three popular methods, which we will review briefly here, are (1) Delaunay Triangulation, (2) advancing front, and (3) grid and quadtree meshing.

Delaunay triangulation generates a unique set of triangulation from an array of input points. This triangulation must satisfy the property that the circumcircle of any triangle must not contain any additional nodes. This triangulation can be performed three ways: 1) with a brute force method, by flipping edges that do not satisfy the circumcircle property, 2) with an incremental node insertion, by adding an additional node where more resolution is needed and re-triangulating, and 3) with subdivision and merging, by splitting the region into multiple sets, triangulating, and then re-merging the sets. The Delaunay triangulation ensures mesh uniqueness [1].

Advancing front [3,4] begins by starting on the domain boundary and incrementally works inwards, changing the mesh front with each new triangulation. The triangles are created by choosing an edge from the front and then either adding a new node to create the triangle or connecting the edge to an existing node. This decision is determined by the user input mesh density. This algorithm continues until the mesh is complete [1].

Grid and quadtree intersection divide the bounding box of the computation domain with structured elements. To take into account the boundary of the object being meshed to, an unstructured mesh is created where the structured mesh intersects the boundary. This mesh can either be left as a hybrid mesh (containing both structured and unstructured elements) or it can be further divided into an unstructured mesh. Care should be taken to ensure there are no hanging nodes [1].

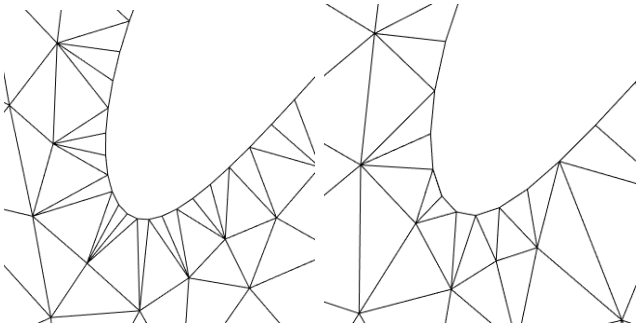
Generally, unstructured meshes provide higher levels of flexibility for objects with complex geometries and local resolution changes can be performed in small regions of the domain. However, the storage of unstructured meshes can be quite large and computationally expensive if running programs

that are $\mathcal{O}(N^2)$ or higher [1].
In this project, we implement Delaunay triangulation with Gmsh [2] to create an unstructured grid around an airfoil (Section 2.1). The mesh undergoes both local (Section 2.3) and uniform refinement (Section 2.4). We discuss our results in Section 3 and conclusions in Section 4.

# 2 Methods

## 2.1 Mesh Generation

The starting coarse mesh was generated in program called Gmsh [2]. To do this, the coordinate data was read into a Matlab script, and then coordinate points and connecting lines were output into a special `.geo` filetype. Original attempts at mesh generation used full airfoil resolution, or in other words every point from the airfoil geometry data was fed into the Gmsh set up. This introduced a number of issues, however. Gmsh uses the points on the geometry as nodes to mate the generated mesh cells to other, program generated nodes off of the geometry. This means that every single point on the airfoil was anchoring a cell. To restrict the coarse mesh to only around 1500 total cells then becomes a significant challenge, as the nodes off of the geometry collapse into a small number of points, but every single node on the airfoil remains. This results in a significant amount of very skewed triangles pointing off of the airfoil, which ends up in floating point errors in the freestream error calculations discussed later.
To fix these issues, the original airfoil resolution was decreased. While not an *ideal* solution it was the quickest and simplest. Different resolutions between $\frac{1}{3}$ and $\frac{1}{5}$ were generated. This means that when reading in the airfoil data, only every third or fifth point were sampled into the final coordinate list. While this does decrease the accuracy of the simulation, the airfoil itself still retained an acceptable amount of its original geometry. Significant geometry-based errors (such as a curved leading edge being made from a low number of sharp, straight lines) only became pronounced at resolutions below $\frac{1}{5}$. In addition, local refinement near the leading and trailing edges will help to accurately resolve the geometry in these regions.



**(a) Poor** **(b) Improved**

**Figure 1: Reducing the mesh resolution improves high aspect ratio cells without significantly compromising geometry**

With an acceptable airfoil resolution the mesh generation followed smoothly. A Delaunay mesh was generated using a box size-field surrounding the airfoil geometry. In the near field, small cell sizes are needed to fully define the airfoil geometry and the resulting gaps between the flap and slat. However, in the extreme far field, such as over 50 or 60 chord lengths away, the cells can be massive as there will be very little flow disruption. The full field mesh is shown below:

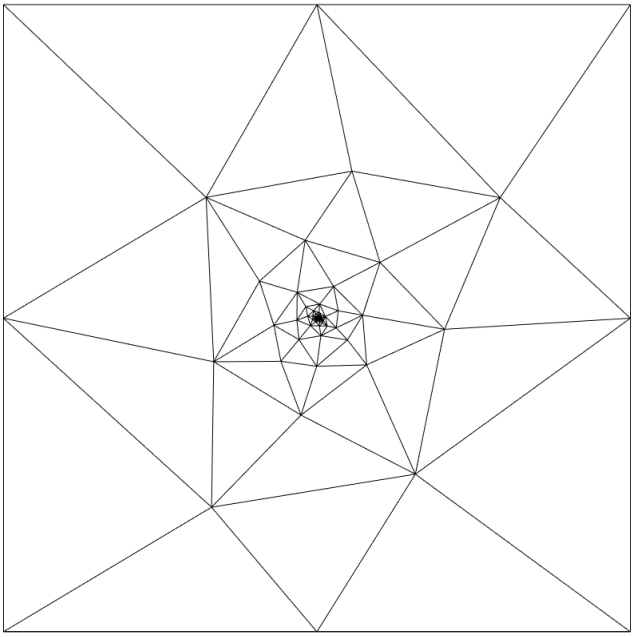


**Figure 2: Full Coarse Mesh**

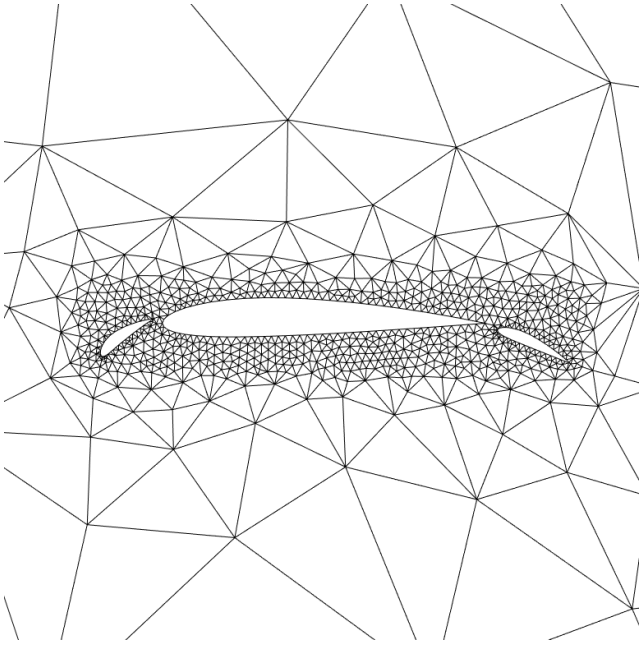


**Figure 3: Near Airfoil Coarse Mesh**

Similarly, the transition from farfield to nearfield mesh is shown in Figure 3. The box size-field is clearly visible.

Finally, the mesh is exported to a `.m` file containing the node coordinates and element connectivity. This is converted into a `.gri` file by finding the boundary edges, separating them by farfield or one of the three airfoils, and recording the nodes associated with the edges in the corresponding boundary section. A `.gri` is then exported containing these node lists, in addition to the the node coordinates, element list, and general mesh information.

#### 2.1.1 Post-Processing

First, the `.gri` file containing the mesh is processed to be in element to node format. The location of each node is also taken from the `.gri` file. The boundaries are also identified from the `.gri` file, where the boundary groups are -1 for the bottom, -2 for the right, -3 for the left, -4 for the top, -5 for the slat, -6 for the main, and -7 for the flap. Then, looping through all the elements, each edge is identified. If that edge is identified twice, it is an interior edge and the interior faces to elements (`I2E`) array is created containing the left element, left local face number, right element, and right local face number, where the left element has the smaller index. If the edge is only identified once, then it is a boundary edge and the boundary faces to elements and boundary groups array (`B2E`) can be created. This contains the element, the local face, and the boundary group identifier.

To calculate the normal on each face, we loop through all the elements and calculate the normal vector. Then, the corresponding normals for each edge are identified when the `I2E` and `B2E` arrays are created.

The area of the elements are calculated using the area of an irregular triangle,

$$s = \frac{e_1 + e_2 + e_3}{2} \tag{1}$$

$$A = \sqrt{s(s - l_{e1})(s - l_{e2})(s - l_{e3})} \tag{2}$$

where Eq. 1 is the semi-perimeter of the triangle and $l_{ei}$ is the length of edge i.

### 2.2 Verification

To verify that the normal vectors and lengths of the interior and boundary faces are correct, for each element we calculate

$$E_e = \sum_{i=1}^{3} \hat{n}_{ei}^{outward} l_{ei} \tag{3}$$

where $\hat{n}_{ei}^{outward}$ is the normal vector and $l_{ei}$, both calculated in Section 2.1.1. This should be near machine precision on each element.

### 2.3 Local Mesh Refinement

It may be desired to decrease the element size at a particular region in the mesh to correctly solve for the flow at that region. This includes leading and trailing edges of airfoils. A local mesh refinement capability helps achieve the desired level of localized reduction in element size. Smoothing of the mesh is one method of improving mesh quality, though quality improvement isn't guaranteed. Local mesh refinement with smoothing function,

`ref_local.m` has been implemented in the present study. The MATLAB file `plotgri.m` (provided by Prof. Fidkowski) was updated and used to read the input `.gri` file. The function developed was tested on three sample `.gri` files before refining the actual airfoil mesh.

#### 2.3.1 Description of Algorithm

**Inputs required**:

- Input `.gri` file
- Coordinates of point about which local refinement is to be done $(x_{lr}, y_{lr})$
- Radius of region around $(x_{lr}, y_{lr})$ that needs to be refined, $r_{lr}$
- Relaxation factor, $\omega$ for smoothing

**Output**: Output `.gri` file with

- Added element and node ids' corresponding to the refined elements
- smoothed mesh for the $\omega$ considered
- Nodes on `Airfoil, Flap` and `Slat` boundary snapped to the geometry splines.

**Procedure for Local Refinement**

1. Copies of nodes and element data matrices from input `.gri` file are made.
2. Find the centroid of all elements and calculate their distance from $(x_{lr}, y_{lr})$. If the calculated distance is less than $r_{lr}$, the element is flagged as **3** (i.e. all edges need to be flagged), else they are flagged as **0**.
3. The table of edges in the input `.gri` file, *edge_info* is generated by considering an edge between all nodes of an element and looping over all elements. The element for which the edge is determined is considered as the element on its left.
4. The duplicate row of nodes in *edge_info* is removed. The element corresponding to the duplicate row of an edge is considered as the element on its right
5. The pair of nodes for each edge having no duplicate row is compared with the list of nodes for each boundary. If both nodes are on a particular boundary, the entry corresponding to the right element of the edge is instead taken to be negative of the boundary group id.
6. A copy of *edge_info* is generated and named *edge_info_boundary*. Only the boundary edges are retained in this table.
7. The node corresponding to the midpoint of each boundary edge is identified and added to the existing array of nodes on a boundary.
8. The elements for each row in *edge_info* are checked if they are flagged as '3' for refinement.
    - If it is a boundary edge and the interior element is flagged for refinement, the edge is flagged **1** for refinement.
    - If it is an interior edge and both bounding elements are flagged for refinement, the edge is flagged **1** for refinement.
    - If it is an interior edge and only one of the bounding elements is flagged as 3, the edge is flagged **1** for refinement AND the the refinement flag of the other element is increased by 1. This ensures that no hanging nodes are generated due to local refinement process.
    - *edge_info* row format = [$node_1$ $node_2$ $element_{left}$ $element_{right}$ *edge-refinement-flag*]
9. For elements flagged '3' for refinement, the coordinates of the edge midpoints are calculated and stored as new nodes in the copied node data matrix.
10. For each element flagged '3' for refinement, four new elements are generated using the vertices and edge midpoints. The new elements are stored in the copied element data matrix.
11. For each element flagged '1' for refinement

(a) The edge marked for refinement and the node opposite to this edge are identified

(b) Two new elements are generated using identified node and midpoint node of identified edge

The new elements are stored in the copied element data matrix.

12. For each element flagged '2' for refinement

(a) The two edges marked for refinement are identified

(b) The vertex node common to the two edges marked for refinement are identified (say $n_1$). Let the remaining nodes at vertices of the element be $n_2$ and $n_3$

(c) Three new elements are generated as:

- both midpoint nodes of the identified edges and $n_1$
- both midpoint nodes of the identified edges and either $n_2$ or $n_3$
- $n_2$, $n_3$ and either of the two midpoint nodes

The new elements are stored in the copied element data matrix.

13. The elements marked from refinement are removed from the copied element data matrix, as new elements have been generated in their replacement.

**Procedure for Smoothing**

1. Copies of *edge_info* and copied node data matrix (which contains the newly-defined nodes) are made.

2. All boundary edges and those not refined are removed from the copied edge data matrix. A matrix of the remaining nodes is made.

3. The newly-defined nodes on the boundary faces are identified and removed from the copied new-node data matrix.

4. The list of nodes identified for smoothing in Step 2 and 3, as well as their x and y coordinates are compiled into a new matrix, *nn_presmooth*

5. A copy of *nn_presmooth*, *nn_smooth* is made

6. For each node $i$ in *nn_presmooth* having location $\vec{x_i}$, the location of the smoothened node, $\vec{x_i}'$ is determined using the equation below ($N_i$ = total number of nodes involved in refining - 1):

$$\vec{x_i}' = (1-\omega)\vec{x_i} + \frac{\omega}{N_i}\sum_{j=N_i}\vec{x_j} \tag{4}$$

$\vec{x_i}'$ is stored in *nn_smooth*

7. The coordinates of the smoothened node is updated in the original node data matrix

8. All the newly-defined nodes on the `Airfoil`, `Flap` and `Slat` boundaries are projected on their respective splines using the `nspline.m` function. This function finds the closest interval of spline breakpoints to a node, and then finds the closest point on that cubic spline interval to the input node. The input node is then moved to this closest point, correctly fitting the airfoil geometry.

The updated node data matrix, boundary node matrix and element-to-note mapping matrix is used to write the output `.gri` file.

## 2.4 Uniform Refinement

The capability to uniformly refine a mesh is an asset for carrying out grid independence studies as well as for refining meshes where significant evolution of flowfiled occurs in the entire domain, such as boundary layer formation in initial regions of a pipe where the domain does not sufficeint length where the flow is fully developed. Uniform mesh refinement function, `ref_uniform.m` has been implemented in the present study. The MATLAB file `plotgri.m` (provided by Prof. Fidkowski) was updated and used to read the input `.gri` file. The function developed was tested on three sample `.gri` files before refining the actual airfoil mesh.

#### 2.4.1 Description of Algorithm

**Inputs required**:

- Input `.gri` file

**Output**: Output `.gri` file with nodes on `Airfoil, Flap` and `Slat` boundary snapped to the geometry splines.

**Procedure for Uniform Refinement**

1. For each element in the input `.gri` file, three additional nodes were defined by calculating the coordinates of the midpoint of all its edges and stored in a matrix for all new nodes.

2. Duplicate node coordinates were removed from the matrix in step 1, and the resulting matrix was added to the node coordinate list in the input `.gri` file to get the new list of all nodes.

3. The table of edges in the input `.gri` file, *edge_info* is generated by considering an edge between all nodes of an element and looping over all elements. The element for which the edge is determined is considered as the element on its left.

4. The duplicate row of nodes in *edge_info* is removed. The element corresponding to the duplicate row of an edge is considered as the element on its right

5. The pair of nodes for each edge having no duplicate row is compared with the list of nodes for each boundary. If both nodes are on a particular boundary, the entry corresponding to the right element of the edge is instead taken to be negative of the boundary group id.

6. A copy of *edge_info* is generated and named *edge_info_boundary*. Only the boundary edges are retained in this table.

7. The node corresponding to the midpoint of each boundary edge is identified and added to the existing array of nodes on a boundary.

8. Corresponding to each element having corner nodes $n_1$,$n_2$ and $n_3$ and edge midpoint nodes $m_{12}$, $m_{23}$ and $m_{31}$ in the input `.gri` file, four new elements are defined as follows:

   - $n_1$, $m_{12}$ and $m_{31}$
   - $n_2$, $m_{23}$ and $m_{12}$
   - $n_3$, $m_{31}$ and $m_{23}$
   - $m_{23}$, $m_{12}$ and $m_{31}$

   All the new elements are stored in a separate matrix.

9. All the newly-defined nodes on the `Airfoil, Flap` and `Slat` boundaries are projected on their respective splines using the `nspline.m` function.

10. The updated list of node coordinates and nodes on boundary faces as well as list of new elements are used to write the output `.gri` file

# 3 Results

## 3.1 Task 1

A coarse mesh with 1554 cells was generated using `Gmsh` [2] which is an open source 3D finite element mesh generator. The mesh nodes in the vicinity of airfoil boundary did not align with the given airfoil splines Figure 3 . The data file for the mesh was generated in both `.geo` and `.m` format which was further converted to `.gri` using a `MATLAB` script.

## 3.2 Task 2

Applying the post-processing described in Section 2.1.1 on the `test.gri`, provided by Prof. Fidowlski, we get the following matrices. Eq. 5 is the `I2E` array, Eq. 6 is the `B2E` array, Eq. 7 is the `In` array, Eq. 8 is the `Bn` array, and Eq. 9 is the `A` array.

$$\begin{bmatrix} 1 & 1 & 2 & 2 \end{bmatrix} \tag{5}$$

$$\begin{bmatrix} 1 & 3 & -1 \\ 1 & 2 & -4 \\ 2 & 3 & -2 \\ 2 & 1 & -3 \end{bmatrix} \tag{6}$$

$$\begin{bmatrix} 0.7071 & 0.7071 \end{bmatrix} \tag{7}$$

$$\begin{bmatrix} 0 & -1 \\ -1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{8}$$

$$\begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \tag{9}$$

### 3.3 Task 3

The verification (as described in Section 2.2) was applied to `test.gri`. The maximum error on each element of this mesh was 0. Then the verification was applied to the coarse mesh from Section 2.1. The maximum error on each element was 0 within numerical precision everywhere. This indicates that the normals are pointed in correct directions and that the vector sum is 0.

### 3.4 Task 4

Local refinement was carried out at the leading and trailing edge of the main airfoil, flap and slat. A radius of 0.04 was taken for identifying the elements to be refined at the slat leading edge (LE) and the flap trailing edge (TE). A radius of 0.06 was taken for identifying the elements to be refined in the gaps between the slat TE-airfoil LE and airfoil TE-flap LE. For smoothing, an $\omega$ of 0.1 was considered. Local refinement was also carried out with an $\omega$ of 0.25, shown for slat LE in Figure 4. As observed in the figure, the smoothing capability presently implemented was leading to highly anisotropic elements in the region of refinement for higher values of $\omega$. Thus the $\omega$ of 0.1 was finalized for local refinement for the present study. Figure 5, 6, 7 and 8 compare the mesh before and after local refinement and smoothing (considering $\omega$=0.1) at the different locations considered for local refinement. Prior to refinement, the mesh had 1554 elements. Post local refinement, the element count in the mesh increased to 2084 elements. The locally refined mesh was checked with the verification (as described in Section 2.2) and has zero error, within machine precision, on each element.

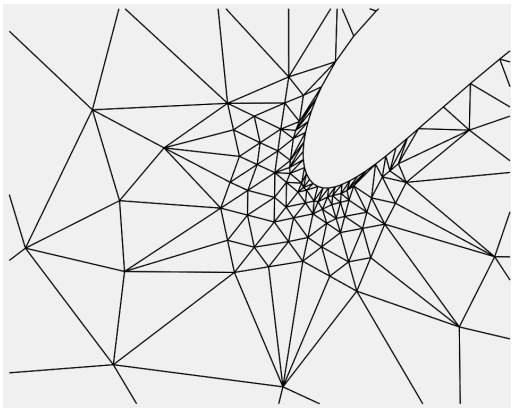
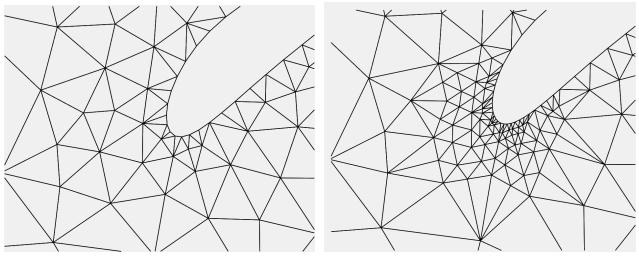
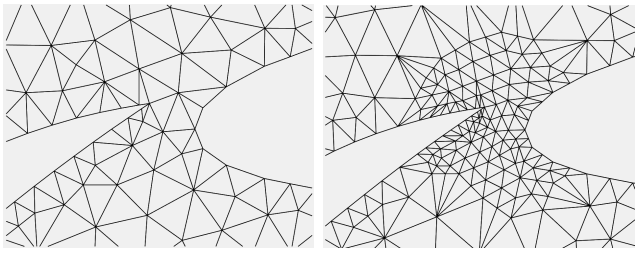


**Figure 4: Change in mesh around leading edge of slat due to local refinement and smoothing ($\omega$=0.25)**



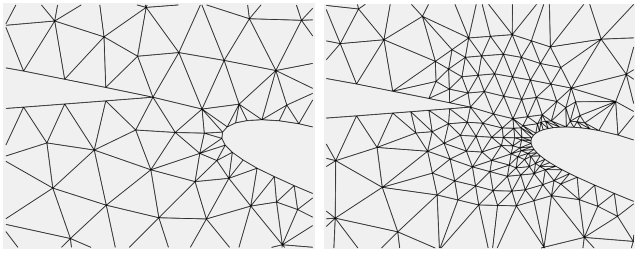
**(a) Before** **(b) After**

**Figure 5: Change in mesh around leading edge of slat due to local refinement and smoothing**



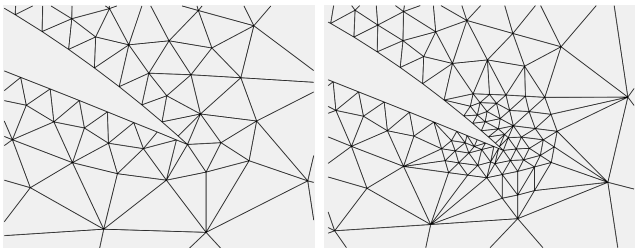
**(a) Before** **(b) After**

**Figure 6: Change in mesh around trailing edge of slat and leading edge of main airfoil due to local refinement and smoothing**



(a) Before (b) After

**Figure 7: Change in mesh around trailing edge of main airfoil and leading edge of slat due to local refinement and smoothing**



(a) Before (b) After

**Figure 8: Change in mesh around trailing edge of flap due to local refinement and smoothing**

### 3.5 Task 5

The `.gri` file corresponding to the locally refined mesh obtained in Task 4 was used as the initial input for carrying out three consecutive uniform refinements of the mesh. Table 1 gives the element count of the mesh for each uniform refinement cycle. 9 shows the mesh of the entire domain after each cycle of uniform refinement. 9 shows the close-up of the mesh generated at the flap-airfoil-slat. Figures 11- 14 show the close-up of the uniform refinement of the locally-refined mesh for the first two uniform refinement iterations.

**Table 1: Element Count of Mesh after Uniform Refinement Cycle**

| Mesh | # elements |
|---|---|
| Locally Refined Mesh | 2084 |
| After $1^{st}$ Uniform Ref. Iteration | 8336 |
| After $2^{nd}$ Uniform Ref. Iteration | 33344 |
| After $3^{rd}$ Uniform Ref. Iteration | 133376 |

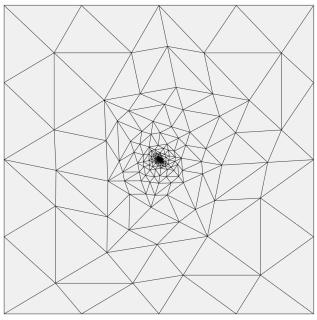
All three uniform refinements of this mesh were run with the verification (described in Section 2.2) and all have zero error, within machine precision, on each element.

## 4 Conclusions

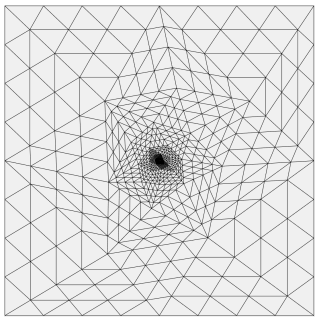
In this project, we generated a mesh for an airfoil with a main component, slat, and flap. First, we created a coarse mesh using `Gmsh` [2], which contains 1554 elements. These nodes were clustered near the airfoil and contained a square, far-field boundary at ±100 (see Section 2.1).

Then, we implemented local mesh refinement of the coarse mesh to create an additional mesh with 2000 elements. The local refinement allows a user to improve resolution in a given region for a specific point and a radius around that location. Each element is then refined and hanging nodes are removed with further refinement. For the local refinement, we smoothed using a weighted average of the nodes to make the mesh more isotropic (see Section 2.4).

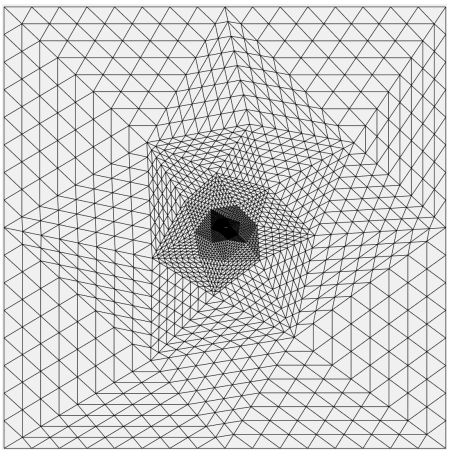
We also implemented uniform refinement of the mesh to create three additional meshes with 8k,



(a) $1^{st}$ uniform refinement

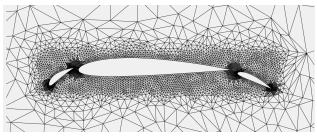


(b) $2^{nd}$ uniform refinement

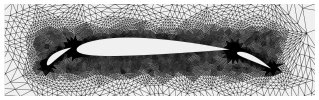


(c) $3^{rd}$ uniform refinement

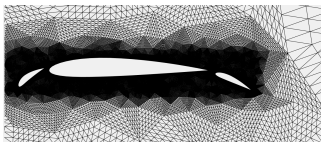
**Figure 9: Change in overall mesh after different cycles of uniform refinement**



(a) $1^{st}$ uniform refinement

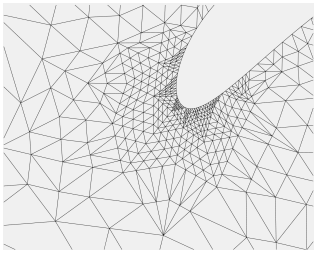


(b) $2^{nd}$ uniform refinement

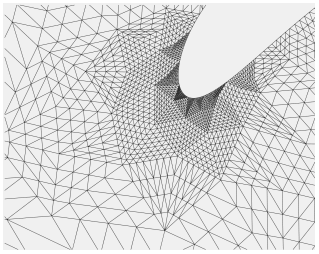


(c) $3^{rd}$ uniform refinement

**Figure 10: Change in mesh around slat-main airfoil-flap after different cycles of uniform refinement**

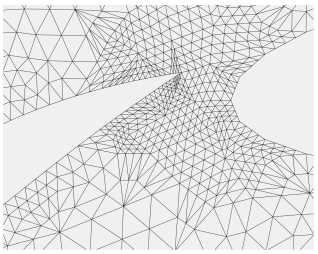


(a) $1^{st}$ uniform refinement

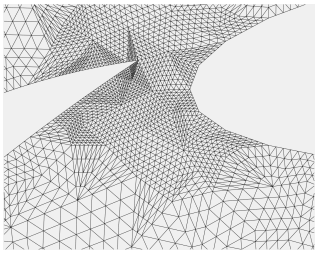


(b) $2^{nd}$ uniform refinement

**Figure 11: Change in locally refined mesh around leading edge of slat due to uniform refinement**

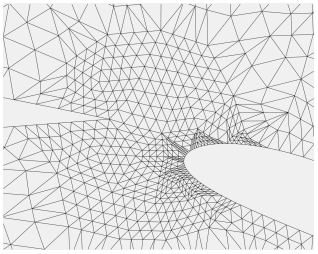


(a) $1^{st}$ uniform refinement

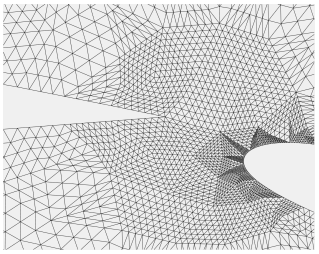


(b) $2^{nd}$ uniform refinement

**Figure 12: Change in locally refined mesh between trailing edge of slat and leading edge of main airfoil due to uniform refinement**

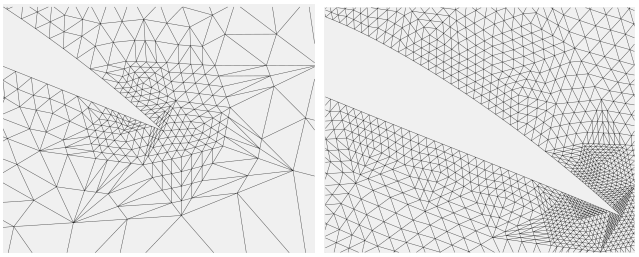


(a) $1^{st}$ uniform refinement



(b) $2^{nd}$ uniform refinement

**Figure 13: Change in locally refined mesh between trailing edge of main airfoil and leading edge of flap due to uniform refinement**



**(a) $1^{st}$ uniform refinement** **(b) $2^{nd}$ uniform refinement**

**Figure 14: Change in locally refined mesh at trailing edge of flap due to uniform refinement**

32k, and 128k elements. For the uniform refinement, each element was split into four subelements through edge bisection. For the curved edges, the nodes are snapped to the cubic spline function describing the airfoil boundary (see Section 2.3).

For these meshes, we verified that the error on each element is near machine precision, using the normal vectors and lengths of each edge (see Sections 2.1.1 & 2.2). All codes were written to be below $O(N^2)$.

Overall, the generation of this mesh with the different levels of refinement is important for resolving flows near the airfoil boundary. The further refinement added, especially near the trailing edge, would improve the accuracy of the flow.

# 5 Effort Breakdown

- Elizabeth (EW) worked on the coding of Task #2 and #3 and the verification of the meshes and wrote about those sections. EW also wrote the abstract, introduction, and conclusion. EW took meeting notes and ensured that meetings were held at reasonable intervals throughout the project developement.

- Moon (MBH) developed the code for Task #4 and #5 (except node projection to spline) and wrote about the same in the present report.

- Ben (BB) worked on generating splines, projecting nodes onto the splines, and mesh generation. He also helped write, edit, and compile the final report and code.

- Vishwa (VMT) worked on and developed a version of local refinement Task #4. He developed a slightly inefficient code for snapping the nodes to spline for `test.gri` and a coarse `main.txt` mesh. VMT helped debug the codes and contributed to the report.

- Tom (TG) learned Gmsh and developed the coarse mesh files and wrote about said section. Unfortunately an illness and subsequent hospitalization limited my overall contribution, however this will be made up for on the following projects.

**Table 2: Effort percentages.**

| | BB | TG | MBH | VMT | EW |
|---|---|---|---|---|---|
| development | 20 | 20 | 20 | 20 | 20 |
| coding | 15 | 15 | 40 | 15 | 15 |
| managing | 15 | 15 | 15 | 15 | 40 |
| report | 20 | 20 | 20 | 20 | 20 |

# References

[1] Krzysztof Fidkowski. Mesh generation, Jan 2023.

[2] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, September 2009.

[3] R. Lohner. Extensions and improvements of the advancing front grid generation technique. *Communications in Numerical Methods in Engineering*, 12(10):683–702, October 1996.

[4] J. Peraire, M. Vahdati, K. Morgan, and O. C. Zienkiewicz. Adaptive Remeshing for Compressible Flow Computations. *Journal of Computational Physics*, 72(2):449–466, October 1987.