AE 623 Project 1: Mesh Generation

January 30, 2023

1 Introduction

The objective of this project is to generate an unstructured triangular mesh for a three-element airfoil. The geometry of the airfoil can be seen in Figure 1.

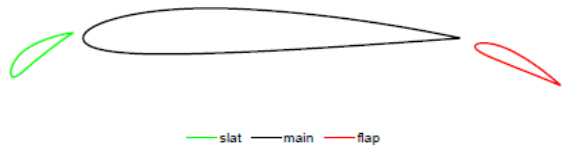


Figure 1: Geometry of the three-element airfoil

The origin of the coordinate system will be set at the leading edge of the main element, which has a unit chord, c = 1. The fairfield boundary is a square where $(x,y) \in [-100,100]^2$. The coarse mesh will consist of around 1500 cells, where there are smaller cells close to the airfoil and larger cells in the fairfield. Uniform refinement will be used to generate meshes with 8k, 32k, and 128k elements. Local refinement will be implemented to improve the resolution in important regions like the leading and trailing edges of the airfoil. Finally, verification will be conducted on all meshes to ensure validity and the meshes will be printed to a *.gri* file format.

2 Methods

2.1 Task 1

The first step is to create a coarse mesh that can be refined. This is done using Gmsh, a finite-element mesh generator developed by Christophe Geuzaine and Jean-François Remacle. In this software, the computational fluid domain is encoded into a *.geo* file. The workflow is as follows:

1. Transcribe the coordinates of the airfoil, slat and flap; reducing the number of nodes to reach the target of 1500 elements.
2. Connect the nodes using lines, creating a closed curve.
3. Create farfield corner nodes and connect with lines.
4. Define fluid domain as planar surface and label the four walls and airfoil boundaries into their respective groups.
5. Generate 1D mesh followed by 2D mesh and export into an appropriate format.

The chosen appropriate format in Gmsh was an ABAQUS input file which contains information about: nodes and their coordinates, elements such as lines and what nodes make up these lines and the elements contained within the surface/fluid domain. The objective is to write a script that creates a *.gri* file and doing so from the ABAQUS is a matter of reordering the information. The *.gri* is an in-house file type with the following format:

Listing 1: GRI file format

```
12 14 2 #node count, element count, dim
1 1 #nodes
...
0.67 0.73
4 #number of boundary groups
2 2 Inlet #number of elements, dim, name
2 8 #node numbers
8 3
...
14 1 TriLagrange #nr. of elem, order, elem type
9 10 11
...
```

Once completed, the .gri file may be visualized using an appropriate library such as *Matplotlib.*

2.2 Task 2

Processing the mesh is done in order to generate four matrices: I2N - a mapping from interior faces to elements, B2E - a mapping from boundary faces to elements and boundary groups, In - normal vectors for interior faces and Bn - normal vectors for boundary faces.

The dictionary data type is the key to organizing faces by storing the nodes that make up the face as the key. The approach for this method is shown in Listing 2 and thereafter one should append to the I2N matrix if the reverse face exists; eg. ‘9 10’ is the reverse of ‘10 9’.

Listing 2: Dictionary creation

```
faces = {} #create dictionary
    for i, E2N_ in enumerate(E2N): #iterate
    ↪ elements
        for j, node in enumerate(E2N_): #for
        ↪ each node
            key = E2N_[j] + ' ' + E2N_[(j+1)%3]
            ↪ #save key, e.g. '9 10'
            faces[key[-1]] = [i+1,j+1] #save
            ↪ element nr, local face nr
```

The code iterates through all elements of the mesh and saves all edge combinations to a dictionary called *faces*. Then index *i* and index *j* are saved which represent global element number and local face number respectively.

For reference, the *test.gri* file will be used to illustrate the methodology and it may be seen in Figure 2. The dictionary *faces* takes on the form: {'1 2': [1, 1], '2 3': [1, 2], '3 1': [1, 3], '2 4': [2, 1], '4 3': [2, 2], '3 2': [2, 3]}. The diagonal face consisting of nodes 2 and 3 are repeated but this is desired.

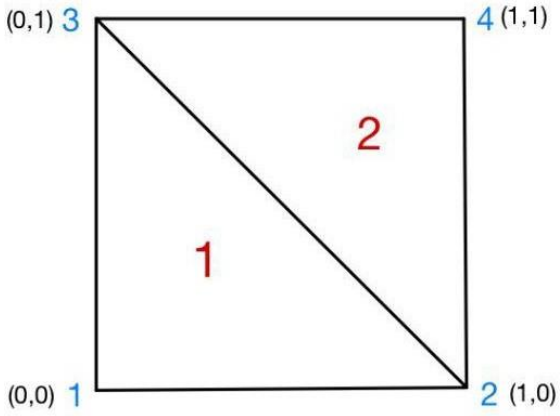


Figure 2: The mesh *test.gri* with node numbers marked in blue and element numbers in red.

This means that faces occurring twice are interior faces while the others are boundaries. Thereafter the interior faces are iterated through, comparing the elements and saving them as *elemL* and *elemR* depending on which has the larger index. The convention is [elemL, faceL, elemR, faceR] where face is the local face number and L/R represent left and right; left is the smaller .

Thereafter the B2E matrix is created, initially when parsing the *.gri* file, the elements or boundary groups are stored in a list. In this case it would look like this: [['1', '2', 'Bottom'], ['1', '2'], ['1', '2', 'Right'], ['2', '4'], ['1', '2', 'Top'], ['4', '3'], ['1', '2', 'Left'], ['3', '1']] and comes directly from the *.gri* file. This is implemented using Listing 3:

Listing 3: B2E creation

```
for bface in walls: #iterate through wall elements from .gri
    if len(bface) == 3: #if containing 3 elements, new boundary group
        name = bface[2] #save name
    else: #store
        inf = faces[bface[0] + ' ' + bface[1]] #elem nr and local face nr
        B2E.append([str(inf[0]),str(inf[1]) ... ,name])
```

Which saves element number, local face number and boundary group. Thereafter the normal vectors of each face are computed by iterating over the faces and determining the change in x and y coordinates; see Listing 4.

Listing 4: Computing normal vectors

```
normalL = []
for face in I2E: #for each face
    elemL = int(face[0])
    faceL = int(face[1])
    nodesL = [E2N[elemL-1][faceL-1], E2N[elemL-1][faceL%3]] #node numbers that build face
    dx = float(C[int(nodesL[1])-1][0]) - float(C[int(nodesL[0])-1][0])
    dy = float(C[int(nodesL[1])-1][1]) - float(C[int(nodesL[0])-1][1])
    normalL.append([-dy/length,dx/length])
```

This process is exactly the same for boundary edges, however instead of using I2E, the matrix B2E is used. Lastly, the area is computed by iterating through E2N which uses a generic formula for the area of a triangle given the coordinates of the three points; see Listing 5.

Listing 5: Element area calculation

```
area = []
for elem in E2N: #for each element
    for node in elem: #for each node
        xycord = []
        for n in C[int(node)-1]:
            xycord.append(float(n)) #save coords of each node
area.append(0.5*abs(xycord[0]*xycord[3] + xycord[2]*xycord[5] + xycord[4]*xycord[1] - xycord[1]*xycord[2] - xycord[3]*xycord[4] - xycord[5]*xycord[0])) #compute area

```

2.3 Task 3

The mesh verification test can be implemented by using the normal vectors and lengths for all the faces in the domain. In this project, we split the faces into interior and boundary faces. The purpose of mesh verification is to ensure that each element's error hovers around machine precision. Once we have the **In** and **Bn** matrices the mesh verification test can be done by looping over the faces in the mesh. The following equation will be used to calculate the maximum magnitude of the error for each element, E_e :

$$E_e = \sum_{i=1}^{3} \vec{n}_{ei}^{\,outward} \, l_{ei}, \qquad (1)$$

Implementing it in a function as follows where the inputs are In and Bn:

Listing 6: Mesh verification

```
def mesh_verification(E2N, NL, mapp):
for elem in range(len(E2N)):
    sumx = 0; sumy = 0
    # mapp, dictionary store face number in each element
    for m in mapp[elem+1]: # loop over each element
        get x and y normal and length from NL[0],[1] and [2] respectively
        if m > 0: add x*l and y*l to sumx and sumy respectively
        else: subtract x*l and y*l from sumx and sumy respectively
    E_n.append([sumx,sumy])
return max(E_n)
```

In order to achieve the result, we create a dictionary to store the number of elements as a key and the number of 3 adjacent faces as a value. We also noted whether each face's normal vector points into or out of that element. By marking the direction of the normal vector, we can easily identify if we need to add the result to or subtract the result from the running total on the adjacent elements. To clarify, if the normal vector points away from the element (the focused element is considered a left element to the adjacent element), the calculated value on that face is added to the element and subtracted from the adjacent element.

Otherwise, it will be subtracted from the element and added to the adjacent element.

2.4 Task 4

We implement local refinement to improve the mesh resolution near the trailing and leading edge of the airfoil. First, we call a function that loops over all the cells and finds its centroid and stores it in a N x 2 matrix, where the column represents the x and y coordinates respectively. We calculate the centroid using the formula give below:

$$Centroid\ of\ elem,\ i = \left(\frac{x_{i,n}+x_{i,n}+x_{i,n}}{3}, \frac{y_{i,n}+y_{i,n}+y_{i,n}}{3}\right) \quad (2)$$

Then, we loop over all the elements and find the distance between the element's centroid and user specified coordinate. If the distance between them is less than or equal to user specified radius, then that element will be flagged for refinement. Afterwards, the element number is stored in a matrix of size N x 1. The following algorithm shows how to implement the above task in a function:

Listing 7: List of flagged elements creation

```
def flag(E2N, user-specified x,y and r):
    #find the centroid of the triangle using the formula given and store it as the matrix "centroid" of size Nx2 (N - number of elements)
    #flagging the centroids
    for i in range(len(E2N)):
    x2, y2 = centroid[i]
    r_prime = find the distance between (x2,y2) and (x,y)
    if r_prime <= r and i not in flag:
        flag.append(i)
    return flag #Size Nx1 containing flagged elements
```

Now, we can loop over the flagged elements, and find the midpoints for each side, these will be our new nodes. Every time we create a new midpoint we assign the node number as: total number of nodes +1; and append its coordinate to the Cord matrix and according to the convention described below, we append the node number to the E2N matrix.

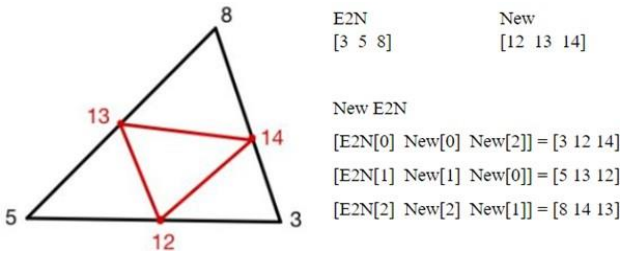


Figure 3: Local refinement for a 3 edges flagged element

It is evident, in doing so we will be adding four new elements to E2N, however, we need to remove the node number corresponding to the bigger element, but we can't just remove it as it will change the size of the E2N and will affect the ordering. Thus, when we visit the element that is being refined, we store its element number in a matrix "mark". At the end of the function, we mark every node number corresponding to the element numbers in the matrix "mark" as FALSE and remove it from the E2N thus not affecting the ordering while the algorithm is still running.

Another problem is when we visit an element adjacent to an element that has been refined, it will share an edge and there already exists a mid-point node. Thus, we need to check if there already exists a node between that element's face, if not then only create a new node. We can do that by creating a dictionary where the key values are the node number of the edge or face and its value being the new node number created between them. Thus, every time we create a new node, we update the dictionary and every time before creating a new node, we check if that edge has already been refined.

Listing 8: Local refinement algorithm and Visited dictionary creation

```
def local(Cord, E2N, I2E, flags):
    visited = {}; n_nodes = len(cord)
    ↪ #initialize
    #looping over the flag elements
    for k in flags:
        find the node numbers of the element and
        ↪ its corresponding coordinates
        newnodes = [0, 0 ,0]
        #looping over the sides
        for j in range(3):
            mids = [node1, node2, midpoint x, y]
            if face in visited: #already refined
                newnodes[j] = value in
                ↪ dictionary
            else:
                update cord matrix, add nodes of
                ↪ the face in dictionary as
                ↪ key and new node number as
                ↪ the value
                newnodes[j] = n_nodes; n_nodes
                ↪ += 1
        update E2N acc. to convention shown in
        ↪ fig
        #element needs to be removed from E2N
        mark.append(i)
```

Now, we need to refine the elements which are adjacent to flagged elements, these elements will fall into one these categories:

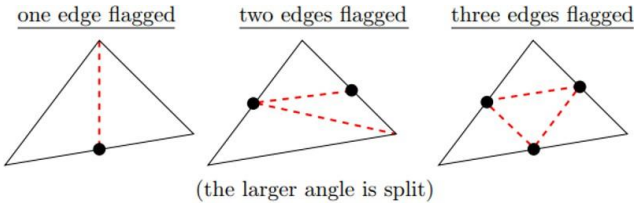


Figure 4: Local refinement implementation for each type of element

We can find the adjacent elements by using the I2E matrix, and from the same matrix we can also find which side it is and nodes for that side. However, first we will make a dictionary that will store the element number as a key and for its value it will store the local face number.

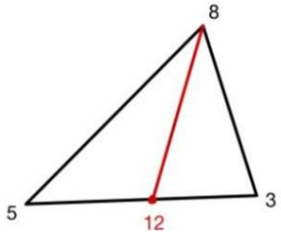
Implementing it in the algorithm in Listing 9.

Listing 9: Flagging adj. Elements's faces

```
mapp = {} #initializing dictionary
generate an empty dictionary with the key as an
↪ element number
for face in I2E:
    #if E_L is flagged then E_R needs to be
    ↪ refined
    if face[0] in flags and not face[2] in
    ↪ flags:
            mapp[face[2]].append(face[3])
    #if E_R is flagged then E_L needs to be
    ↪ refined
    elif face[2] in flags and not face[0] in
    ↪ flags:
        mapp[face[0]].append(face[1])
    #removing elements that are not adjacent to
    ↪ flagged elements
    mapp = {k: v for k, v in mapp.items() if v}
```

We are appending the local face number as the same element can have multiple sides that need to be refined. We can use this fact to find how to refine the corresponding element, by checking its length. If the length is 1 then we use one edge flagged method and if 2 then two edge methods. We do not need to create a new node as it shares the edge that has already been refined in the previous step. From the local face number, we can find its corresponding node numbers. Then, we can look up the dictionary that we previously made which stores the node numbers of the edge to find the node number of the midpoint of the edge. Then we update the E2N matrix and update the bigger element' row in E2N as FALSE as what we did in the previous step.

To update the E2N for a single face flagged method, it depends on the local face number. So, let's assume the element given in the figure, then we can implement the following algorithm.



Local Face 1 [12 5(1) 8(2)] E2N [3 5 8] [12 8(2) 3(0)]

Local Face 2 [12 5(2) 8(0)] E2N [8 3 5] [12 8(0) 3(1)]

Local Face 3 [12 5(0) 8(1)] E2N [5 8 3] [12 8(1) 3(2)]

(index) - represents the index in E2N

Figure 5: Convention for local refinement for a single edge flagged element

Listing 10: Convention for a single edge flagged local refinement

```
if face == 1:          #if face is 1
    E2N.append([node,E2N[1],E2N[2]])
    E2N.append([node,E2N[2],E2N[0]])
elif face == 2:        #if face is 2
    E2N.append([node,E2N[2],E2N[0]])
    E2N.append([node,E2N[0],E2N[1]])
else:                  #if face is 3
    E2N.append([node,E2N[0],E2N[1]])
    E2N.append([node,E2N[1],E2N[2]])
```

For two face flagged methods there are some additional steps. When two edges are flagged, it is necessary to first identify which edges are flagged and then add a new middle node to each face. Next, we divide the element by connecting the two newly created nodes. Then, the angle at the nodes of the remaining face that has not been flagged is determined. The two new triangles will be formed by joining the node with the largest angle to the node opposite it. Thus, the two flagged edge elements will be separated and replaced with three new elements. Implementing two-sided flagged refinement in an algorithm:

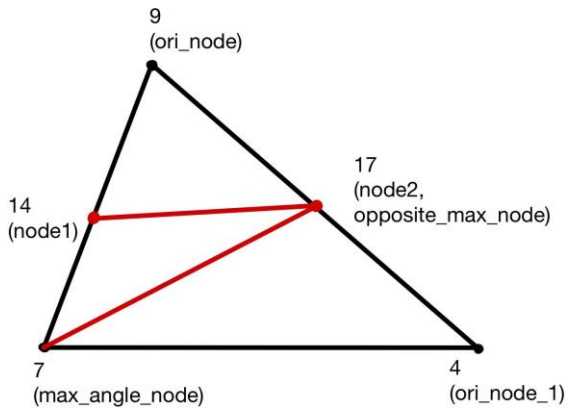


Figure 6: Convention for local refinement for a 2 edges flagged element

Listing 11: Local refinement for two edges flagged

```
if number of flagged == 2:
    #then 2 of the element's faces are flagged
    # Get new node number by using 2 old nodes
    ↪ in E2N to be a key
    node1 = visited[key1]; node2 = visited[key2]
    ori_node = original node btw newly generated
    ↪ nodes
    angle_list = angles at each node
    Get node number that has the maximum angle
    #node that is opposite to the max angle node
    face_opposite = [x for x in E2N[elem] if x
    ↪ != max_angle_node]
    opp_max_node = visited[face_opposite]
    max_node = original node that has the
    ↪ maximum angle
    node_1 = the remaining original node
    E2N.append([node1,ori_node,node2])
    E2N.append([node2,max__node,node1])
    E2N.append([op_max_node,node_1,max_node])
```

We can then smooth the affected nodes by using the formula given below.

$$\vec{x}_i' = (1 - \omega)\,\vec{x}_i + \frac{\omega}{|N(i)|}\sum_{j \in N(i)} \vec{x}_j \qquad (3)$$

However, we first need to find the node neighbors of each node. We can implement the simple algorithm to find the node neighbors and store it in a dictionary where the key is the node number and the value is its node neighbors. One only needs to run smoothing on interior nodes thus we need to update our B2E matrix, to identify the new edges that were created previously. We changed the B2E, into a dictionary where the key is nodes of the edge on the boundary and value being in the first column be 0 everywhere and boundary group name in the second column. Then while running the local refinement function every time when a new node is created, we will change the 0 to be the new node number if it is a boundary face. Like following code:

Listing 12: Identifying new nodes on all the boundaries

```
#dict_b is a dictionary with the key being the nodes of the element's face
key = str(nodes[j]) + ' ' + str(nodes[(j+1)%3])
if key in dict_b: #then it is a boundary face
    dict_b[key][0] = new node number on that face number
```

Now, we can use this fact to update our dictionary using the following algorithm to include the new boundary faces.

Listing 13: Creating new edges on the boundary and updating B2E

```
def bound_dict_f(dict_b):
bound = []   #intializing
for i in dict_b:
    if dict_b[str(i)][0] > 0:
    #then there is a new node on that face
    #node1 and node2 are value of node in key
    bound.append([node1, new node, bgroup])
    bound.append([new node, node2, bgroup])
    else:
    bound.append([node1, node2, bgroup])
    return bound_dict
```

Afterwards we use this dictionary to make a vector of size N x 1, where N is the number of nodes on the boundary. Then, we can loop over all the nodes except the nodes on the boundary using the dictionary created above, by using the following algorithm:

Listing 14: Smoothing implementation

```
for i in range(no of smoothing iteration):
for elm in range(len(Cord)):
#temp - nodes on the boundary
    if not elm in temp:
    #update the node's coordinate
    #smoothing function just use the formula to compute the new x,y coordinates
        C[elm] = smoothing(elm,Cord,neigh,0.8)
def smoothing(N(node number),C,neigh,w):
    xi = C[N]; temp = [0,0]
    eq1 = [(1-w)*x for x in xi]
    for i in neigh[N+1]:
        temp = [x+y for x, y in zip(temp,C[i-1])]
    eq2 = [(w / len(neigh[N+1]))*x for x in temp]
    return [x+y for x, y in zip(eq1,eq2)]
```

After we perform smoothing on the mesh nodes affected by refinement, the boundary nodes that have been overlooked will be snapped to the true geometry of the splines. We create the spline by performing two-dimensional splines. Next, we projected the new boundary nodes after the local refinement to the splines. We implemented this by finding the node that gives the minimum distance between the boundary node and the true geometry node. Finally, we snap the node by changing the node's coordinates to the true geometry node. This way, the new boundary nodes created after the local refinement are moved to the real coordinate. Listing below shows the sample code to implement snapping.

Listing 15: Snapping implementation

```
def closest_point(true_spline, node):
    # Computing distance btw the coordinates
    distances = cdist(node, true_spline)
    # Find the index of the point in spline.
    index_array = np.argmin(distances, axis=1)
    # Output: Closest true coordinate
    return true_spline[index_array]
for i in range(No of Nodes):
    if boundary == 'Main' or 'Flap' or 'Slat':
        # Get new coordinate by calling the function
        new_coord = closest_point(TS, coord[i])
        # Update coordinates matrix
        coord[i] = new_coord
```

2.5 Task 5

Uniform refinement is the same as local refinement, however this time we loop over all the elements instead of only flagged elements. (The same algorithm was used just the "for loop" now loops over all the elements).

3 Results

3.1 Task 1

The unrefined coarse mesh is presented in Figure 7 and Figure 8 where the x-coordinate is the horizontal axis and the y-coordinate is the vertical axis.

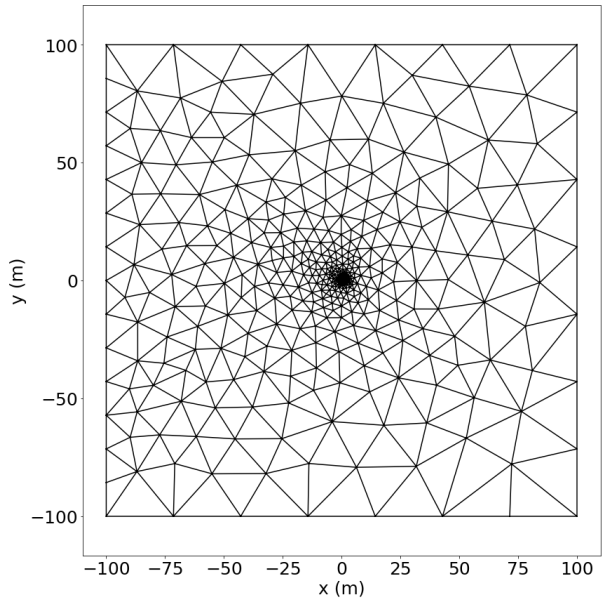


Figure 7: Coarse mesh generated from Gmsh

This mesh contains 805 nodes and 1499 elements where the density increases significantly toward the center of the domain where the airfoil, slat and flap lies; there is no forced refinement at the leading and trailing edges. The number of nodes at the boundaries have been prescribed so that there are 15 nodes at the inlet and 8 nodes on the other boundaries.

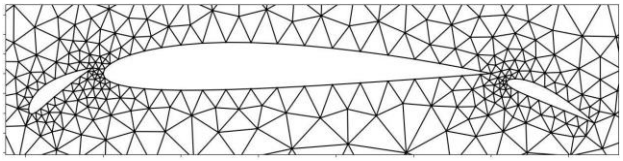


Figure 8: Zoomed in view of the coarse mesh around the constituent airfoil parts.

The benefit of step 1 in section 2.1, reducing the number of nodes on the airfoil surface becomes clear. Without this, it would be nearly impossible to reach the desired target of <1500 elements. In Gmsh the option to "extend element sizes from boundary" was also selected.

3.2 Task 2

For the *bgroup*, **1**, **2**, **3**, **4** refer to 'Bottom', 'Right', 'Top' and 'Left', respectively.

The **I2E** matrix is: **[[1, 2, 2, 3]]**.

The **B2E** matrix is: **[[1,1,1], [2,1,2], [2, 2, 3], [1, 3,4]]**.

The **In** matrix is: **[[0.707, 0.707]].**

The **Bn** matrix is: **[[0.0, -1.0], [1.0, 0.0], [0.0, 1.0], [-1.0, 0.0]].**

The **Area** matrix is: **[0.5, 0.5]**

3.3 Task 3

The maximum magnitude of the error on each element of the **test mesh is equal to zero**, and the maximum magnitude of the error over the entire domain for the coarse airfoil mesh is equal to $3.55^{-15}e$. The magnitude is close to machine precision; for double-precision operations it should be approximately 10^{-16}. Since the magnitude of the error is close to machine precision, one assumes that the mesh is valid.

3.4 Task 4

Applying local refinement at the trailing and leading edge for slat, flap and main airfoils are done according to Table 1.

Table 1: Local refinement locations

Location	x-coordinate	y-coordinate	r
Slat L	-0.23	-0.13	0.1
Slat T	-0.026	0.014	0.02
Main L	0	0	0.07
Main T	1	0	0.04
Flap L	1.041	-0.0194	0.04
Flap T	1.2653	-0.1246	0.1

After local refinement the mesh consists of 1094 nodes and 2043 elements. See Figure 9 for critical areas before refinement and Figure 10 afterwards.

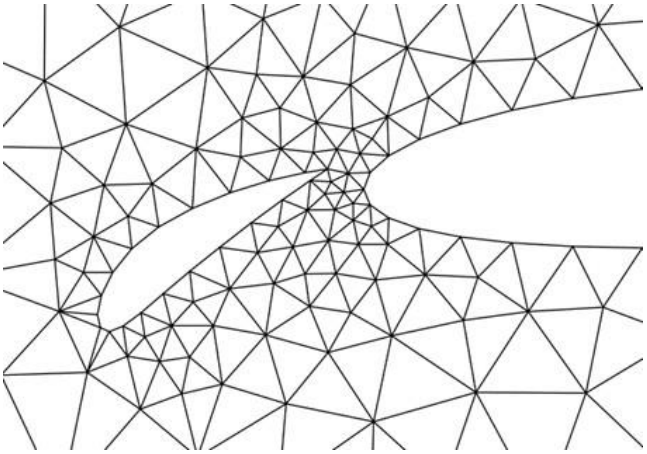


Figure 9: Trailing edge of slat and leading edge of main airfoil before local refinement.

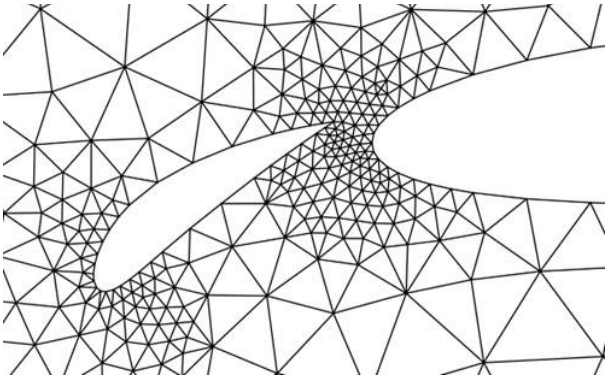
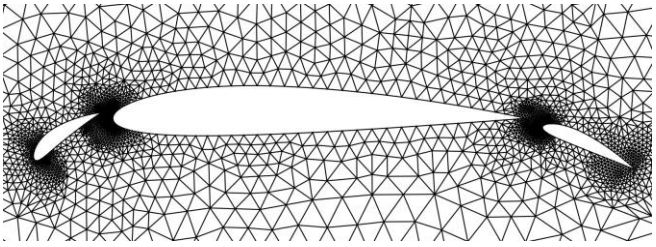


Figure 10: Trailing edge of slat and leading edge of main airfoil after local refinement.

3.5 Task 5

The figures below show pictures of the meshes after 1st, 2nd ,and 3rd uniform refinement implementations. We also used the mesh verification test to ensure that our algorithms did not produce any additional errors in addition to the machine precision error.

After one iteration of uniform refinement the mesh consists of 4242 nodes and 8172 elements.



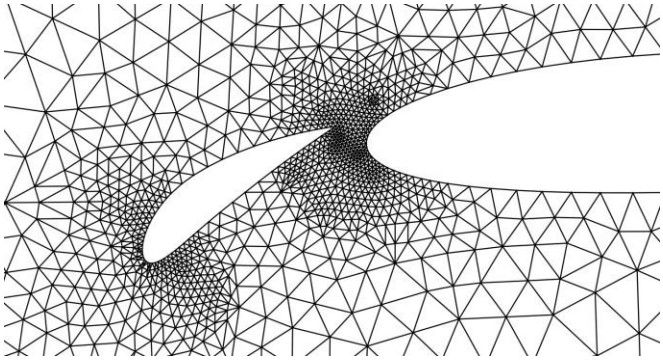


Figure 11: Mesh after one iteration of uniform refinement a) whole geometry b) close-up

After two iterations of uniform refinement the mesh consists of 16667 nodes and 32688 elements.

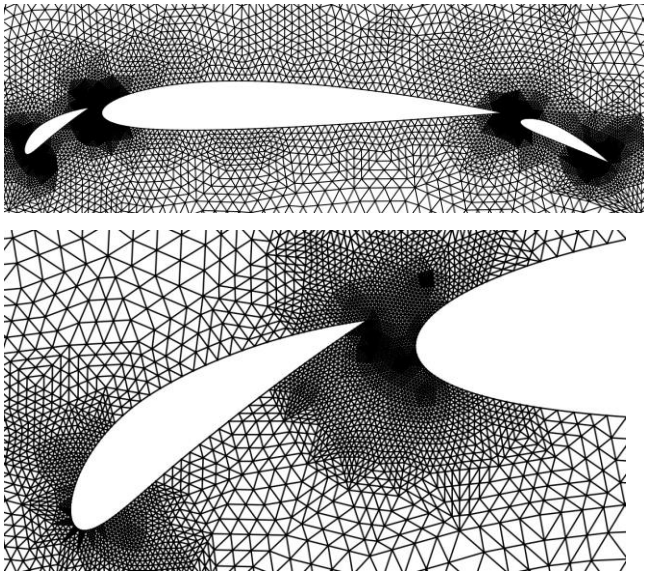
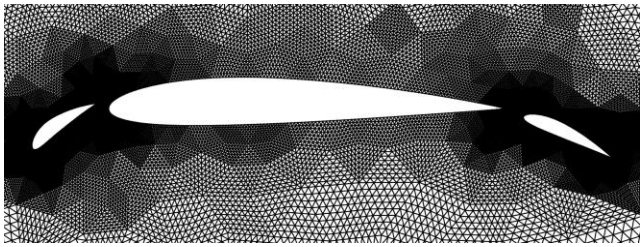


Figure 12: Mesh after two iteration of uniform refinement a) whole geometry b) close-up

After three iterations of uniform refinement the mesh consists of 66033 nodes and 130752 elements.



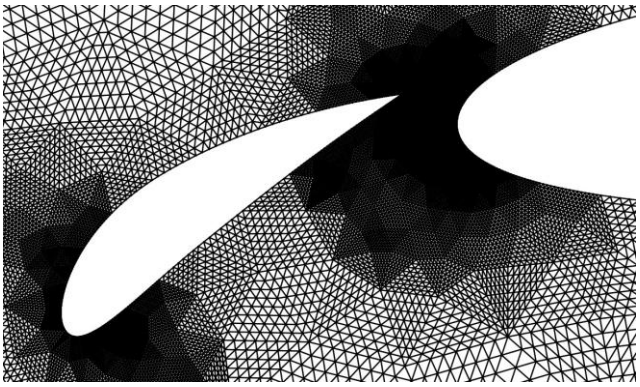


Figure 13: Mesh after three iteration of uniform refinement a) whole geometry b) close-up

Mesh verification of the three uniform mesh refinements are presented in Table 2. The maximum error is close to machine precision and hence the mesh verification test passes for all cases.

Table 2: Mesh verification

Nr. of Elements	Maximum Error
8k	1.43e-15
32k	3.81e-15
128k	5.67e-15

4 Conclusions

Triangular unstructured meshes were generated around a 3-component airfoil using Gmsh. Thereafter the mesh was locally refined using user-specified coordinates and radii. Following local refinement is smoothing which amends elements with high aspect ratio or skewness. Lastly, in order to improve the overall resolution of the mesh, uniform refinement was implemented.

Special care was taken in order to preserve boundary groups which are used in later steps such as mesh verification and writing the processed mesh back into a *.gri* file.

All algorithms were implemented without excessive nesting to preserve O(N) complexity. Improvements to the running time would include switching to a compiled language such as C++, however due to time constraints the knowledge of the language was lacking. The initial mesh consisted of 1499 elements while the final mesh contained 130752 elements.