Theoretical and Computational Acoustics

Nejma Chahine

October 16, 2023

1 Introduction

Our objective with this initial stage is to establish a suitable environment for further studies in computational acoustics. We will be numerically solving Partial Differential Equations (PDEs) on the mesh that we will develop throughout this report. Initially, we will focus on rectangular meshes with square elements. From there, we will extend our analysis to differently shaped meshes with triangular elements, which are more suitable for solving PDEs.

We will arrange the questions in a way that structures the project, focusing on computing the quality of the final mesh first and then proceeding inversely to the chronological order to get the full picture. from now on we will utilise pointers to indicate elements and nodes and we will for the rest of the report use:

1. `elem2nodes`: A numpy array of node IDs used in each element sequentially.
2. `p_elem2nodes`: A numpy array of pointers indicating where, for each element, the list of associated nodes begins in `elem2nodes`.
3. `node_coords`: A numpy matrix storing the coordinates of each node sequentially from lower ID to higher ID.

2 Part 1: Quality Analysis

2.1 Question 1: Aspect Ratio Calculation

The aspect ratio is a quality metric for elements, ranging from zero for extremely misshapen elements to one for regular shapes (the value zero is impossible to obtain for any concave shape). The formula depends on the number of edges of the elements. For triangles, the aspect ratio is given by:

$$Q = \alpha \cdot \frac{h_{max}}{\rho}$$

where h_{max} is the longest edge of the triangle, ρ is the radius of the inscribed circle $\left(\rho = \frac{Area}{\frac{1}{2}perimeter}\right)$, and $\alpha = \frac{\sqrt{3}}{6}$.

For quadrangles, the formula is:

$$Q = 1.0 - \frac{\sum_{j=1}^{4} |e_j e_{(j+1)\%4}|}{4}$$

where $\mathbf{e_j}$ is the elementary vector supporting each side of the quadrangle.

We have initiated the function `compute_aspect_ratio_of_element(node_coords, p_elem2nodes, elem2nodes, type="triangle")` in `quality_analysis.py`. This function calculates the aspect ratio of different elements, assuming that we have a well-defined mesh. Further insights into mesh formation will be explored in later sections. For triangles, we utilize another function, `properties_triangle( coord1, coord2, coord3)`, which computes properties based on the coordinates of each node composing the triangle, returning a dictionary with keys "area," "perimeter," and "hmax."

2.2 Question 2: Edge Length Factor Calculation

Similarly to the first question, we will compute the edge length factor for different elements. This factor represents the ratio of the shortest edge length to the medium edge length.
The function `compute_edge_length_factor_of_element(node_coords, p_elem2nodes, elem2nodes)` in `quality_analysis.py` handles this computation.

2.3 Question 4: Visual Representation of Quality Criteria

Based on the results from the previous two questions, we can create visual representations of the assessed quality criteria, specifically aspect ratio and edge length factor. We achieve this through the function `analysis(p_elem2nodes, elem2nodes, node_coords, element_type="triangle")`. This function performs the necessary computations and generates graphical representations for the two types of elements considered: quadrangles and triangles.

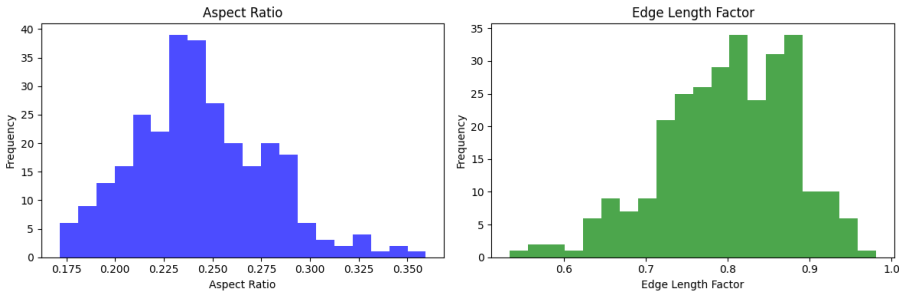

Figure 1: Distribution of the quality factor for a given triangular mesh.

3 Part 2: Creating the Mesh

3.1 Question 6: Splitting a Mesh Composed of Quadrangles into Triangles

In this part of the project, we are using built-in functions provided by Mr. Magoules in `solutions.py`, specifically the function `_set_quadmesh(`x_{min}`, `x_{max}`, `y_{min}`, `y_{max}`, nelemsx, nelemsy)`. This function returns an appropriate mesh represented by `elem2node`, `p_elem2node`, and `node_coords`, as defined in the introduction.

As the name suggests, this function constructs a rectangular mesh within the domain defined by $\{x, y | x_{\max} > x > x_{\min}$ and $y_{\max} > y > y_{\min}\}$ with *nelemsx* and *nelemsy* nodes in each direction (horizontal and vertical).

An important aspect to consider while constructing the mesh is the order of the points in `p_elem2node`. This order verifies a crucial aspect in which nodes are ordered starting from the smallest node pointer and increasing to the next smallest pointer among neighboring nodes.
example: in the case of a square with nodes $0, 1, 2, 3$, where nodes 0 and 2 are on opposite sides, the proper ordering would be $0, 1, 3, 2$.

Since triangular elements are better suited for acoustic calculations, allowing for smaller errors and more precise results, we utilize the mesh produced previously to generate an equivalent mesh but with triangular elements. For simplicity, and given that the mesh is already in the shape of a rectangle, we divide each rectangular element into two rectangular triangles.

This division is achieved by doubling the length of `p_elem2nodes` and rearranging the elements in `elem2nodes`. Depending on the orientation of the triangles we desire, we divide the rectangle.
Assuming the nodes of the rectangle are in the `Nodes` list, there are two possible divisions:

Division 1:

$$elem1 = Nodes[0], Nodes[1], Nodes[2]$$
$$elem2 = Nodes[0], Nodes[2], Nodes[3]$$

Division 2:

$$elem1 = Nodes[0], Nodes[1], Nodes[3]$$
$$elem2 = Nodes[1], Nodes[2], Nodes[3]$$

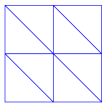

Figure 2: triangles in division 1

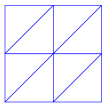

Figure 3: triangles in division 2

It's important to note that in each of these divisions, care is taken to maintain a positive angle of rotation to preserve the integrity of structures used in other parts of the code, as required for subsequent questions. The function used for this purpose is 'split_quad_tri(p_elem2nodes, elem2nodes)', which is found in 'creation_peigne.py'.

3.2 Question 3: Shifting Internal Nodes in the Mesh

To create a function that shifts the coordinates of the internal nodes of a mesh without modifying the nodes located on the boundary, we can use the `disturbed_nodes(node_coords, p_elem2nodes, elem2nodes)` function. This function takes the coordinates of the mesh nodes as input and shifts each node based on a controlled disturbance.

The constraint here is to only modify the internal nodes. Internal nodes have a specific property: they are limited to having 4 corresponding elements for the quadrilateral case and 6 for the triangular case, following the laws of geometry for squares and right triangles. It is essential to understand that this property is a result of the way the mesh was initially constructed and is not an inherent property of the triangles themselves. As you may have noticed, we began with more regular shapes and are moving towards irregular shapes, allowing us some flexibility in controlling the node positions.

We iteratively identify internal nodes by checking the number of connected elements and disturb their original positions.

To achieve this, we can use the `disturbed_nodes(node_coords, p_elem2nodes, elem2nodes)` function, which takes the node coordinates and translates each node by a factor of `step * random_percent(0, 0.5) * e /` $\|\mathbf{e}\|$ where $\mathbf{e}$ is a random vector with components `(random(0,1) *` $\mathbf{e_x}$ `+ random(0,1) *` e_y`)`. e_x `and` $\mathbf{e_y}$ `are the horizontal and vertical vectors,step represents the original distance between two nodes.`

The result is a mesh with disturbed internal nodes, as shown in the following image:

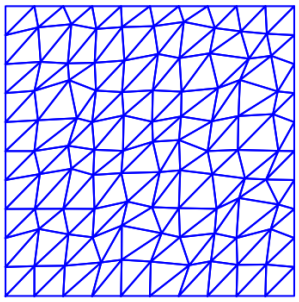

Figure 4: Disturbed Mesh

4 Part 3: Creating the Fractal

4.1 Question 7: Mesh Manipulation Functions

In this part, we need to create functions for adding and removing elements and nodes in the mesh, which are essential for creating a fractal. These tools allow us to manipulate the mesh we have created in the previous sections.

4.1.1 Adding an Element

To add an element to the mesh, we use the function add_elem_to_mesh(node_coords, p_elem2nodes, elem2nodes, elemid2nodes). This function extends the elem2nodes array with the new set of nodes used in elemid2nodes, while ensuring the correct order of the nodes. We also extend p_elem2nodes with a new integer indicating the last element added to elem2nodes.

4.1.2 Removing an Element

To remove an element, we use the function remove_elem_to_mesh(node_coords, p_elem2nodes, elem2nodes, elemid). This function takes elemid, the ID of the element to remove. It starts by removing the associated part in elem2nodes. Since pointers are stored sequentially (the pointer for element n+1 is the pointer for element n plus the number of nodes in element n), we adjust the pointers for elements with IDs higher than elemid by subtracting the number of nodes in the eliminated element.

4.1.3 Adding a Node

Adding a node involves concatenating its coordinates to node_coords. However, it's important to be careful, as the node may not be added in line with previously defined nodes and could have an ID quite different from its surrounding nodes. Using it to create elements can result in unordered element IDs.

4.1.4 Removing a Node

Removing a node requires a more complex approach, as nodes are not as intuitively designed as elements. We use the function remove_node_to_mesh (node_coords, p_elem2nodes, elem2nodes, nodeid).
Here's how it works:
1. Select elements that contain the node to be removed and delete them sequentially. To ensure that the IDs of the remaining elements are not affected, we process the elements in decreasing order of IDs.
2. Delete the associated coordinates of the node from node_coords.
3. Restore appropriate IDs for the remaining nodes, especially in elem2nodes, by subtracting one from the IDs of nodes greater than nodeid. This ensures that the mesh retains its integrity by maintaining appropriate elements.

Exemple: here we took off a node in the lowest row which deleted the two adjacent elements and we added a separate element highlited on the top left corner

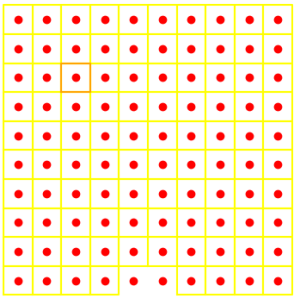

Figure 5: transformed mesh

4.2 Question 8: Generating Peaks and Valleys

In this question, we prioritize the conservation of volume rather than the explicit fractal shape. Our goal is to create a comb-like structure with randomly generated peaks and valleys on the easternmost surface, starting from a simple rectangle.

To create this structure, we use the function creation_maillage(hauteur=10, lar_moyenne=10, amplitude_max=8, amplitude_min=2, epaisseur=1, type='triangle', order=False). Here's how the function works:

- hauteur is the number of elements stacked on the y-axis.
- lar_moyenne is the average width for a rectangle with equivalent volume to our structure.
- amplitude_max is the maximum height of the peaks and valleys.
- amplitude_min is the minimum height.
- epaisseur describes the width of the peaks.
- type defines the shape of the elements, either 'triangle' or 'quad'.
- order is a boolean variable that dictates if we need shifts in the position of the nodes.

To create the mesh, we follow these steps:

1. Generate a rectangular mesh with an appropriate height and a width double the lar_moyenne value.
2. Delete elements from higher IDs to lower IDs, ensuring an alternation between peaks and valleys. For peaks, we have a length of lar_moyenne + Rnumber, and for valleys, we have a length of lar_moyenne + Rnumber, where Rnumber is randomly generated between amplitude_max and amplitude_min. Importantly, Rnumber is only updated after a valley to maintain a constant volume. This operation mimics the idea of taking material from valleys to create peaks next to them.
3. Clean up the mesh by removing non-essential nodes, which are solitary nodes that have no connected elements.
4. Transform the square elements into triangles and shift nodes if needed, depending on the parameters of the function.

The result of this function is node_coords, p_elem2nodes, and elem2nodes. This approach allows us to generate a mesh with a comb-like structure while maintaining a certain level of order in element IDs.

we chose this approach with consecutive peaks and valeys to preserve a constant variance on all peaks , different methods have to get notably higher variances for the last peaks in the case for a mesh with constant volume

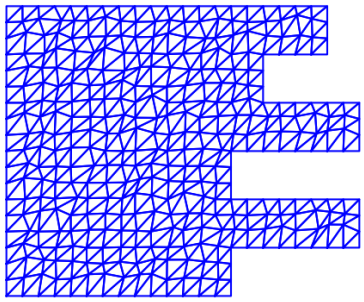

Figure 6: comb-like mesh with peaks of width 3

5 Part 4: Visualization

5.1 Question 5: computing barycenters

sequentially we compute the barycenter of each element

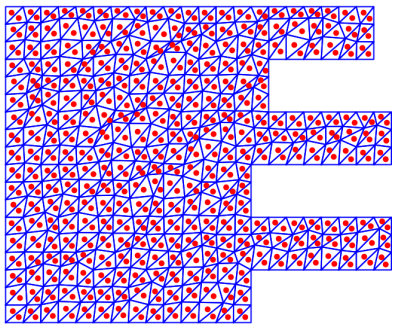

Figure 7: comb-like mesh with barycenters shown

6 Conclusion

In this study, we embarked on a journey through the realms of theoretical and computational acoustics, aiming to create an environment for further investigations in this field. Our exploration led us to delve into mesh generation, quality assessment, and geometric transformations, all of which are crucial components of computational acoustics.

As we reflect on our journey, we recognize that there is much more to explore and discover in this multifaceted field. Our research paves the way for future investigations, from refining mesh generation techniques to exploring the real-world applications of acoustics. The path forward is both exciting and challenging, and we eagerly anticipate the next steps in this fascinating journey.