Optimal recombination algorithms for generating quad-dominant meshes

Undergraduate Thesis

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Aditya Jaiswal
ID No. 2021A4TS2201P

Under the supervision of:

Dr. Aravind Balan
&
Dr. Aneesh A.M

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS
December 2024

Declaration of Authorship

I, Aditya Jaiswal, declare that this Undergraduate Thesis titled, 'Optimal recombination algorithms for generating quad-dominant meshes' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 12/12/2024

Certificate

This is to certify that the thesis entitled, "*Optimal recombination algorithms for generating quad-dominant meshes*" and submitted by Aditya Jaiswal ID No. 2021A4TS2201P in partial fulfillment of the requirements of BITS F421T Thesis embodies the work done by him under my supervision.

Supervisor
Dr. Aravind Balan
Faculty,
Indian Institute of Science (IISc)
Date: 16/12/2024

Co-Supervisor
Dr. Aneesh A.M
Asst. Professor,
BITS-Pilani Pilani Campus
Date:

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS

Abstract

Bachelor of Engineering Mechanical Engineering (Hons.)

Optimal recombination algorithms for generating quad-dominant meshes

by Aditya Jaiswal

Simulations in computing, particularly within the realms of Computational Fluid Dynamics (CFD) and structural dynamics, are fundamentally dependent on two critical components: numerical schemes and mesh configurations. Various numerical techniques require different discretization approaches; for example, Lattice Boltzmann Methods (LBM) and Finite Difference Methods (FDM) divide the domain into discrete nodes, where differential equations are solved at each point. In contrast, Finite Element Methods (FEM) and Finite Volume Methods (FVM) utilize cell elements, allowing the transformation of differential equations into integral forms through the application of the Gauss divergence theorem and calculus of variations, respectively, in each set of methods.

Enhancing the accuracy and efficiency of simulations through advanced mesh adaptation strategies, specifically leveraging metric field-based approaches, we arrive at the "most" ideal solution. By deriving a metric field that reflects error estimates, this method of mesh adaptation optimally aligns the mesh with the evolving features of the solution. This adaptive approach enables refined resolution in areas of interest while minimizing computational resource allocation in less critical regions.

This thesis focuses on the recombination of mesh elements - the conversion of a triangular mesh to a hybrid/quad dominant mesh, which is a part of the adaptive mesh refinement process (AMR). The main idea behind metric-field based mesh adaptation is to generate a unit mesh in a prescribed Riemannian metric space, which is equipped with similar notions of length and volume as the Euclidean metric space [3].

After mesh adaptation cycles, we obtain a triangular mesh, used as an input to the recombination program. The recombination program sorts and assigns priority values to elements and their neighbors, which is then followed in the recombination process and we end up with a quad dominant mesh. The entire motivation behind recombination process is to align the mesh and the elements concerning the flow in order to resolve the flow features and, more specifically, is essential in compressible flows.

In this thesis, we first discuss some element sorting parameters that will guide us in converting the triangulations in mesh to quads based on priority orders. Later, we discuss parallelizing the recombination program to increase efficiency.

Acknowledgements

Undertaking this bachelor's thesis has been a massive learning and enjoyable experience for me, and I would like to thank the people for their help and guidance.

First and foremost, I would like to express my sincere gratitude to my research supervisor, Dr. Aravind Balan for his guidance and advice during my research. He constantly encouraged me, guided me throughout the work, and taught me a lot about my research problem and how to be a good researcher overall.

I would like to thank Dr. Aneesh A. M. for co-supervising me on the thesis, for regularly taking updates from me and solving problems with his insightful guidance on my research problem and for believing in me to take up the research problem.

I would like to thank the team, with a special mention to Dipendrasingh Kain for all their experience and support throughout my thesis.

I am grateful to the Computation Laboratory at IISc, Bangalore, and Birla Institute of Technology and Science (BITS) Pilani for providing the opportunity and infrastructure to pursue my bachelor's thesis and experience a great learning and research environment.

I would like to thank my friends for standing by me and making my time here joyful. I am indebted to my family for their unconditional love and support and for their blessings in believing in me to complete this thesis. My parents for all their sacrifices at different stages of their lives to ensure that my brother and I pursue the best education and grow with ample opportunities to which they did not have access. I want to thank them from all my heart for their values, their drive for excellence, and their contributions to date and for the future.

Contents

List of Figures

Listings

Abbreviations

$\mathbf{a}$,$\mathbf{b}$,$\mathbf{u}$,$\mathbf{v}$,$\mathbf{x}$	Vectors or Points
$\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$	Vector Co-ordinates
$\mathbb{R}^3$	Three Dimensional Space
$\mathcal{M}$	Metric field or Metric
$<\mathbf{u},\mathbf{v}>$	Inner Product space
$\mathrm{d}_{\mathcal{M}}(\mathbf{a}, \mathbf{b})$	Distance between points $\mathbf{a}$ & $\mathbf{b}$
Ω	Domain ϵ $\mathbb{R}^2$
$\mathcal{V}(\mathbf{a})$	Vicinity of $\mathbf{a}$

Chapter 1

Introduction

The quality of the mesh used to discretize the domain has a major impact on the accuracy and efficiency of numerical solutions in computer simulations, especially in domains like fluid dynamics and structural analysis. Metric field-based mesh adaptation, which dynamically refines the mesh according to the underlying solution properties, has become one of the ways to do so, and some meshing softwares like Bamg and BL2D in 2D [1] has the capabilities where the user can specify metric fields that can define the orientation, size or shapes and uses the metric approach. This approach uses a metric tensor to determine the appropriate mesh modification based on solution gradients and error estimates. Metric field-based adaptation ensures that computational resources are distributed more efficiently, improving solution accuracy without needlessly raising computational costs by adjusting the mesh to match the important aspects of the solution, such as boundary layers, shock waves, or regions of high gradient.

Adapting the mesh is crucial for capturing and addressing various flow variables that may not be captured by a coarse mesh. A static mesh often fails to resolve and represent certain phenomena like shock waves or boundary layers. Using adaptive techniques, the mesh can be refined in regions where the flow shows intricate/sharp behaviours while being coarsened in areas where the solution is smoother. This selective refinement not only improves the accuracy of the simulation but also saves computational power and resources.

After one mesh-adaptation cycle, we get an adapted mesh that consists of all triangular elements. Mesh recombination, which is a recombination of the mesh elements from tri to quad / quad-tri mixed, is essential to resolve boundary layers and also for accuracy in compressible flow simulations.

Quad elements in the mesh are four-sided polygonal elements. They are commonly used in Computational fluid dynamics (CFD) and Finite element analysis (FEA), which acts as a stencil

and basis for numerical computation. They offer several advantages over triangular elements, making them more suited for certain applications. Some of them are listed below:

1. Higher accuracy when solving for structured flows - Quad elements typically align better with flow features such as direction and geometrical features when used for structured grids. This reduces numerical errors and improves accuracy, specifically when used in problems involving boundary layers or compressible flows.

2. Reduced numerical diffusion - In problems involving transport phenomena, like advection-diffusion or convection, quad elements are usually better than triangular elements, especially in capturing strong gradients or interfaces.

3. Better Aspect ratio control - Quad elements can be readily elongated or aligned in the flow direction and offer a better change in orientation as compared to when working with triangular elements; this is specifically beneficial for resolving anisotropic features such as boundary layers or shock waves.

4. Lesser degrees of freedom - For the same geometry, usually, after mesh generation, a complete quad mesh or quad-dominant mesh (hybrid mesh) requires fewer elements than when compared to a complete triangular mesh. This reduces the number of degrees in the simulation, offering lower memory requirements and also faster computations.

5. Improved mesh quality in certain regions - Quad elements are generally easier to control than triangular elements. This is a necessity in obtaining higher quality meshes where the orientation of elements affects the accuracy of simulation; for example, sometimes triangular elements can be highly skewed or have very "extreme" angles in complex geometries, leading to poor numerical results, for this reason, quad elements can be used which are better at resolving flow features and easier to change the orientation of.

6. Quality metrics -

   - Better and faster convergence in iterative solvers.
   - More stability in time-stepping schemes.
   - When using higher polynomial bases, reduced interpolation errors. [5]

1.1 Metric field

A metric is a structure or space on a manifold that defines distances or angles. A manifold can be defined as a topological space that locally resembles an Euclidean space. The metric field, in our case, is calculated by the fact that the distance between two adjacent points or an edge

length is always an unit. Euclidean and Riemannian metric spaces and their operations are elaborated below. A natural product between $\mathbf{u}$ & $\mathbf{v}$ is defined as; $<\mathbf{u},\mathbf{v}> = \Sigma_{i=1}^{3}\ \mathrm{u}_i\ \mathrm{v}_i$.

1.1.1 Euclidean metric space

Euclidean metric space is a finite vector space where the dot product is defined by symmetrical definite positive tensor $\mathcal{M}$:

$$< \mathbf{u}, \mathbf{v} >_{\mathcal{M}} \ = \ < \mathbf{u}, \mathcal{M}\mathbf{v} > \ \equiv \ \ \mathbf{u}^T \mathcal{M} \mathbf{v} \tag{1.1}$$

In general, the metric $\mathcal{M}$ is a 3×3 matrix which is defined such as:

1. symmetric $\forall$ ($\mathbf{u}$,$\mathbf{v}$)
2. positive $\forall$ $\mathbf{u}$
3. definite $\implies$ if inner product is equal to 0, then, $\mathbf{u} = 0$

It can be inferred now that the metric $\mathcal{M}$ for a Euclidean space is given by I_3, which is the identity matrix in 3 dimensions.

$$\mathcal{M}_{euclidean} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.2}$$

Also, from the above descriptions, it can also be inferred that taking the inner product (of two vectors) defined in Euclidean metric space gives us the normed vector space, which gives the length of a vector and, thus, the distance between two points can be simply defined as $\mathrm{d}_{\mathcal{M}}(\mathbf{a},\mathbf{b}) = \|\mathbf{b} - \mathbf{a}\|_{\mathcal{M}}$.
Also, the above property for the distance between two points does not hold true for Riemannian metric spaces. The most important parameters in a mesh are the length of edges, volumes and angles.

1.1.2 Riemannian metric space

Riemannian metric space can be considered a smooth manifold in which the tangent space $\mathbf{T}_a\mathrm{M}$ at each point $\mathbf{a}$ is a Euclidean metric space. In the context of mesh adaptation however, we do not have the notion of a manifold, so, we think of it as Riemannian metric space which is given as $\mathbf{M} = (\mathcal{M}(\mathbf{x}))_{x \epsilon \Omega}$.

1.2 Geometric interpretation

In $\mathcal{V}(\mathbf{a})$ the set of points at distance ϵ from $\mathbf{a}$ are [4]:

$$\phi_{\mathcal{M}}(\epsilon) = \left\{\mathbf{x} \in \mathcal{V}(\mathbf{a}) \mid (\mathbf{x}-\mathbf{a})^T \mathcal{M}(\mathbf{x}-\mathbf{a}) = \epsilon^2\right\} \tag{1.3}$$

Since $\mathcal{M}$ is a symmetric matrix, it is diagonalizable. Thus, the spectral decomposition theorem states:

1. **Real symmetric matrices** have all real eigenvalues.

   If $\mathcal{M}$ is a real symmetric matrix, then all its eigenvalues are real.

2. There exists an orthonormal matrix R (i.e., $R^T = R^{-1}$) such that $R^T A R$ is a diagonal matrix, where A is the real symmetric matrix. This means that the matrix can be diagonalized via an orthogonal change of basis.

   $$R^T \Lambda R = \mathcal{M} \tag{1.4}$$

   where $\mathcal{M}$ is the Metric and $R^T = R^{-1}$.

3. The columns of R are the eigenvectors of $\mathcal{M}$, and the diagonal entries of the resulting diagonal matrix Λ are the eigenvalues of $\mathcal{M}$.

   $\mathcal{M}\mathbf{v}_i = \lambda_i \mathbf{v}_i$, where $\mathbf{v}_i$ are the eigenvectors of $\mathcal{M}$ and λ_i are the eigenvalues.

We now establish the geometric interpretation from 1.3 using formulations defined below:

Let's consider (**x**-**a**) = **y**

Also from, 1.4, we can write ($\mathbf{y}^T\mathcal{M}\mathbf{y}$, can be thought of scaling $\mathbf{y}$):

$$\mathbf{y}^T\mathcal{M}\mathbf{y} \equiv \mathbf{y}^T R\Lambda R^T\mathbf{y} \implies (R^T\mathbf{y})^T\Lambda(R^T\mathbf{y})$$

Consider, R^T**y** as **z**

$$\text{So, } \mathbf{z}^T A\mathbf{z} \text{ where, } \Lambda = \begin{bmatrix}\lambda_1 & 0\\ 0 & \lambda_2\end{bmatrix}$$

$$\equiv \begin{bmatrix}\mathbf{z}_1 & \mathbf{z}_2\end{bmatrix}\begin{bmatrix}\lambda_1 & 0\\ 0 & \lambda_2\end{bmatrix}\begin{bmatrix}\mathbf{z}_1\\ \mathbf{z}_2\end{bmatrix}$$

$$\implies \lambda_1\mathbf{z}_1^2 + \lambda_2\mathbf{z}_2^2$$

Now, let's replace z with $\tilde{\mathbf{x}}_i$-$\tilde{a}_i$. This step aims to convert from the initial quadratic form to 1.5 in the eigenvectors frame.

$$= \sum_{i=1}^{2} \lambda_i (\tilde{\mathbf{x}}_i - \tilde{a}_i)$$

Also, replacing λ_i with $\frac{1}{h_i^2}$

$$\equiv \sum_{i=1}^{2} \left(\frac{\tilde{\mathbf{x}}_i - \tilde{a}_i}{h_i^2} \right)^2 \tag{1.5}$$

Hence, the final set form of all the points that are in the vicinity of $\tilde{a}$ given by $\tilde{\mathbf{x}}$ at unit distance is;

$$\phi_{\mathcal{M}}(1) = \left\{ \tilde{\mathbf{x}} \in \mathcal{V}(\tilde{\mathbf{a}}) \mid \sum_{i=1}^{2} \left(\frac{\tilde{\mathbf{x}}_i - \tilde{a}_i}{h_i^2} \right)^2 = 1 \right\} \tag{1.6}$$

From the above equations, it can be inferred that it is the equation for the representation of an ellipsoid centred at $\mathbf{a}$ (shown in figure 1.1) with its axis aligned towards the direction of eigenvectors of $\mathcal{M}$ and sizes along the axis are $\mathrm{h}_i = \lambda_i^{-2}$. It is also called as an "unit ball". Figure 1.2 shows how the metric can be used to map from Physical space to Metric

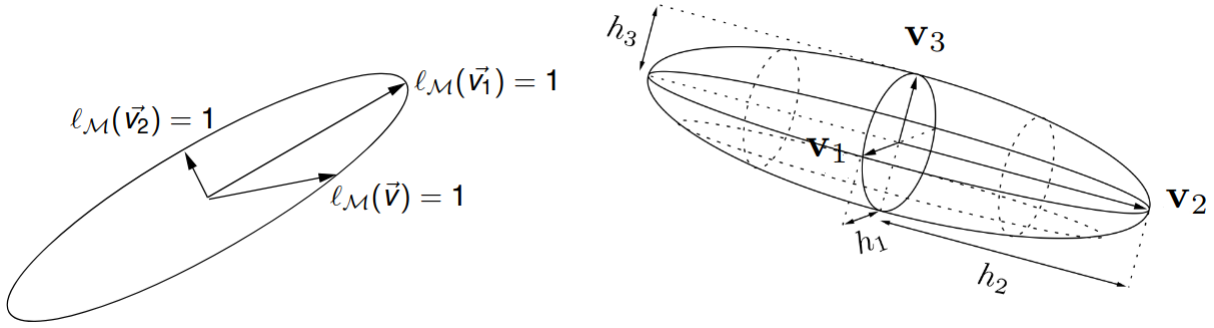


FIGURE 1.1: An ellipsoid or unit ball in $\mathcal{M}$, $\mathbf{v}_i$ are the eigenvectors and the axis direction, whereas, the distance along each axis is h_i.

space and vice-versa. To end things concisely on "mapping", shown below are the set of simplifications:

- $\|\mathbf{x}\|^2 = \mathbf{x}^T\mathbf{x}$
- $\|\mathbf{x}\|^2_{\mathcal{M}} = \mathbf{x}^T\mathcal{M}\mathbf{x} \equiv \left(\mathcal{M}^{1/2T}\mathbf{x}\right)^T \left(\mathcal{M}^{1/2}\mathbf{x}\right)$

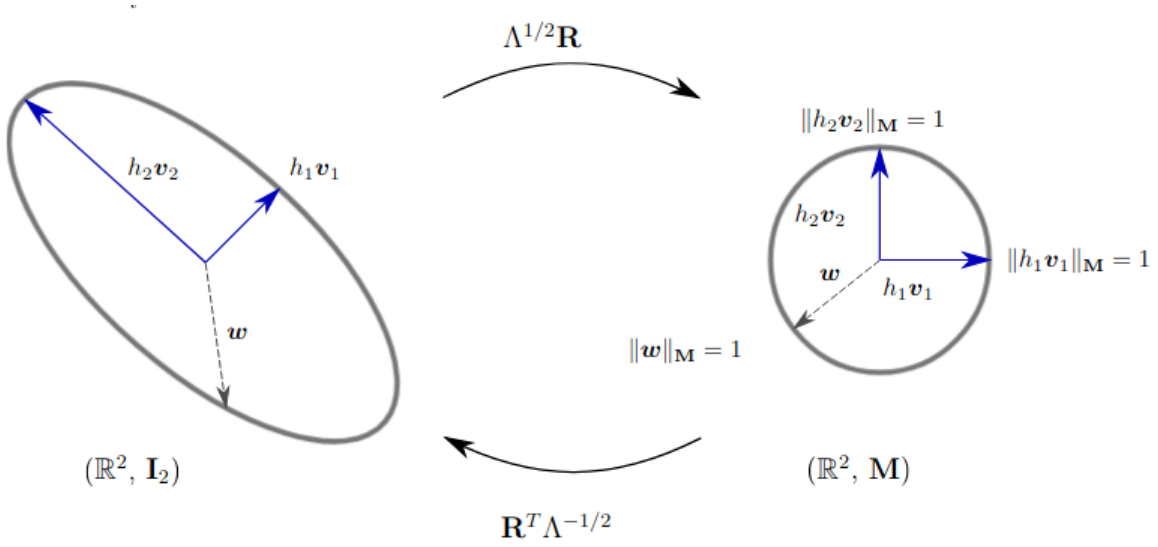


FIGURE 1.2: Mapping of elements from Physical space $\mathcal{I}_2$ to Metric space $\mathcal{M}$

The second point illustrates how the $\mathbf{x}$ is "distorted" in a metric field. Also, from the figure 1.2:

$$\mathcal{M}^{1/2} = R\Lambda^{1/2}R \qquad \text{where} \quad \Lambda^{1/2} = \text{diag}(\lambda_i^{1/2})$$

1.3 Motivation

In computational fluid mechanics, particularly concerning compressible flows and boundary layers, the adaptation and recombination of meshes from triangular to quadrilateral or mixed quad-tri arrangements becomes essential. Boundary layers exhibit strong anisotropic behaviour, with steep velocity gradients perpendicular to the wall and smoother variations along streamlines, requiring meshes aligned with the flow direction. Quadrilateral meshes, especially the elongated varieties, are more efficient in capturing these layers with fewer elements compared to triangular meshes, which often need excessive refinement as the interpolation errors are reduced along with numerical artifacts. *For context, in high Reynolds number flow, the elongated quad elements capture thin boundary layers better and reduce the computational time compared to more dense tri elements.*

Compressible flows feature shocks and strong discontinuities, thus accurate gradient representation is vital. Quadrilateral meshes reduce numerical diffusion and artificial smearing because they can align effectively with sharp features, improving solution accuracy in high-gradient regions like shocks. In addition, solvers optimized for structured meshes tend to show better convergence due to consistent element topology, enhancing numerical stability and computational efficiency. *Also, in shock-capturing schemes, the shock can be*

captured effectively better using quad elements as they can align with shock waves reducing oscillations and artificial smearing.

Starting with triangular meshes allows flexibility in forming complex geometries, but transitioning triangles into quadrilaterals in critical areas maximizes computational resource usage. A hybrid mesh may also be used which is a quad-tri mixed mesh with quads at all important regions, while, tri elements being elsewhere.

Quadrilateral elements yield greater accuracy for each degree of freedom compared to triangles, requiring fewer elements to capture flow characteristics accurately, thus lowering computation time and memory usage. Ultimately, positioning quadrilateral elements with streamline orientations through flow-driven methods diminishes numerical diffusion and improves solution accuracy, rendering them essential for addressing high-shear and advection-dominated phenomena in boundary layers and compressible flows.

1.4 Objective

The objective of this thesis is to explore and advance techniques in mesh generation using triangular elements, with a focus on recombination methods. This work aims to develop and evaluate methods that improve mesh quality by recombining triangular elements to achieve better adaptability and resolution across a wide range of structure types, enhancing the precision of finite element analysis.

A core aspect of the thesis will involve investigating algorithms for the recombination of mesh elements. These algorithms will be applied to both regular and irregular geometries, with the goal of reducing element distortion while maintaining flexibility in meshing complex shapes.

The suggested study will be verified with the use of computer programs like Gmsh and NETGEN. Both these software can be used to visualize as well create mesh. The thesis's ultimate goal is to offer fresh perspectives on mesh optimization that may be used to improve simulation performance in scientific and engineering applications.

Chapter 2

Literature Survey and Data

2.1 Literature Survey

The field of work recombination and adjustment has experienced significant advance, especially through the improvement of ceaseless work systems and metric-based strategies. A noteworthy commitment in this region comes from Loseille and Alauzet [4, 1], who introduced a continuous mesh framework using Riemannian metric spaces achieving anisotropic mesh adaptation. This framework gives complete control over interpolation errors through metric fields, ensuring the best alignment of the elements with solution features. These framework also enables a control over orientation and density offering a complete control over the mesh, offering a robust approach for adaptive mesh refinement, independent of the inital mesh quality.

Within the particular setting of mesh recombination, the Blossom-Quad algorithm proposed by Remacle et al. [7] represents the utilize of graph-theoretical methods to produce high-quality quadrilateral mesh from triangular mesh. By using a minimum-cost perfect-matching algorithm, this strategy guarantees optimized elements and sizes whereas following to initial mesh. As a recombination technique, Blossom-Quad exceeds expectations in mesh recombination techniques. The Blossom-quad algorithm is also used by Gmsh which is a finite elemental mesh generator.

The work of Borouchaki and Frey [2] investigates the integration of anisotropy in the recombination process. This enables the conversion or recombination of meshes according to the solution obtained, tailoring to the specific needs of the simulation.

Applications of these standards are distinctively illustrated within the work of Ghalia et al. [3], who utilize progressed anisotropic work adjustment methods to address the

challenges of boundary layer and turbulence simulations. By integrating specifics of boundary-layer meshing strategies and their corresponding metric fields, this approach resolves high-gradient regions near the walls of geometries. This illustrates the significant importance of adaptive mesh refinement, recombination and mesh smoothing techniques in CFD applications.

In summary, the integration of metric-based strategies and other recombination strategies represent a pivotal advancement in adaptive mesh refinement and recombination techniques. By the alignment of mesh properties with the underlying physical phenomenons, these methods are well suited for the simulations in problems involving boundary layers or compressible flows [6] which require the techniques of mesh recombination. These advancements in meshing techniques highlight the importance of recombination techniques and mesh refinement in improving the accuracy CFD simulations.

Chapter 3

Problem Setup

The problem is divided as follows:

1. Generating mesh for different geometries consisting of triangular elements.
2. Developing a program to sort the triangular elements based on different sorting methods and then recombine them to form quads
3. Statistical comparison of mesh based on parameters like the problem setup, aspect ratio and the number of triangular and quad elements

3.1 Mesh setup

Various mesh files have been tested and visualized in the program. Although the mesh format is interchangeable and different formats can be converted into others, we chose to work with **.vol** extension, which can be read by the **Netgen** software.

A script can also be run to construct the geometry, then mesh it and refine it if needed. For example, the script to generate a quarter-circle with triangular elements is given below.

```
import netgen.geom2d as geom2d

geo = geom2d.SplineGeometry()
p1,p2,p3,p4 = [ geo.AppendPoint(x,y) for x,y in [(0,0), (2,0), (2,2), (0,2)] ]
geo.Append (["line", p1, p2])
geo.Append (["spline3", p2, p3, p4])
geo.Append (["line", p4, p1])

```

```
mesh = geo.GenerateMesh (maxh=0.05)
# mesh = geo.GenerateMesh(maxh=0.1, quad_dominated=True)

mesh.GenerateVolumeMesh()
mesh.Save ("newmesh.vol")
```

LISTING 3.1: Manual mesh generation script example

The above script, after execution, creates a **newmesh.vol**, which contains the mesh information. Figure 3.1 shows the mesh visualised after running the script.

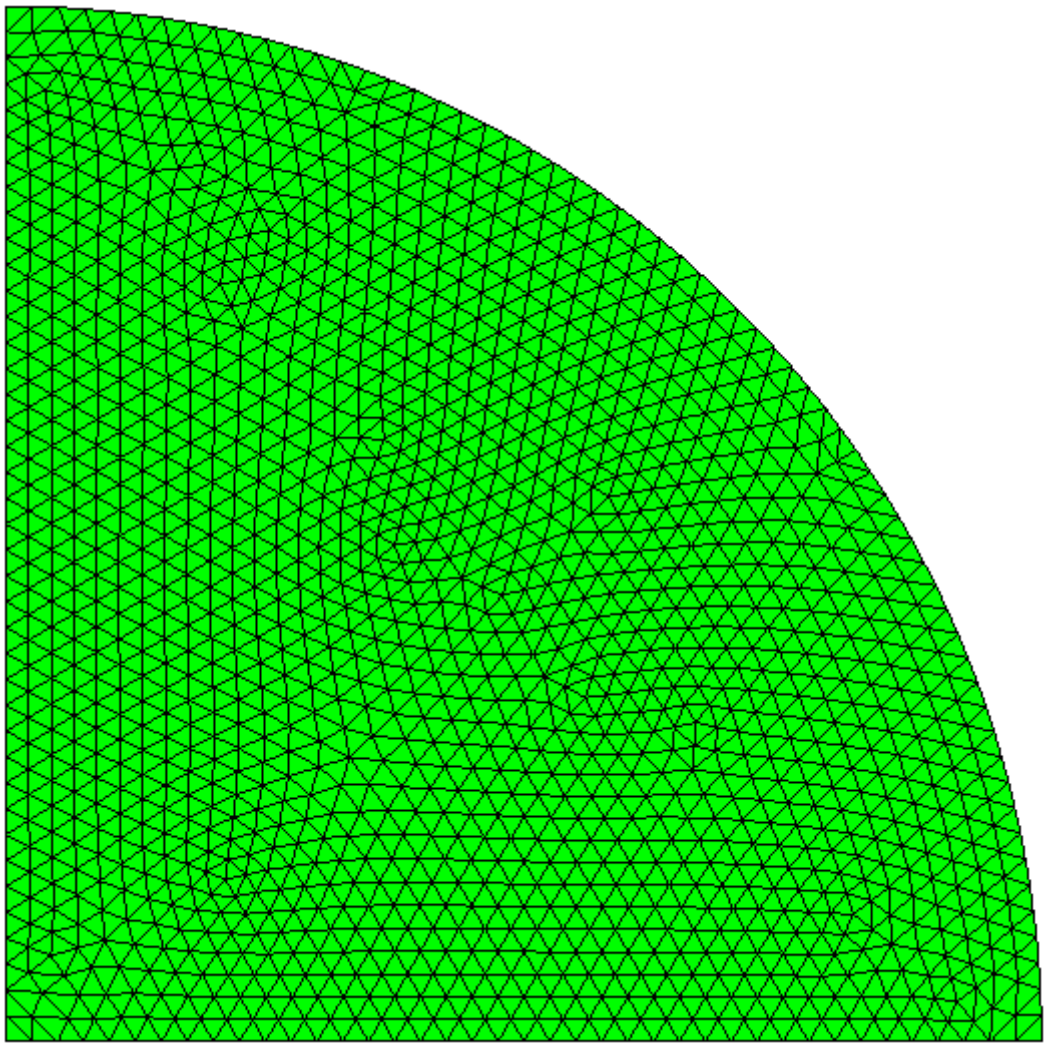


FIGURE 3.1: Quarter circular geometry mesh

The above mesh consists of **2868** triangular elements with a maximal global mesh-size of 0.05.

3.1.1 Different geometries/mesh

A few other meshes that are being considered in the study are given below from figure 3.1 to figure 3.7:

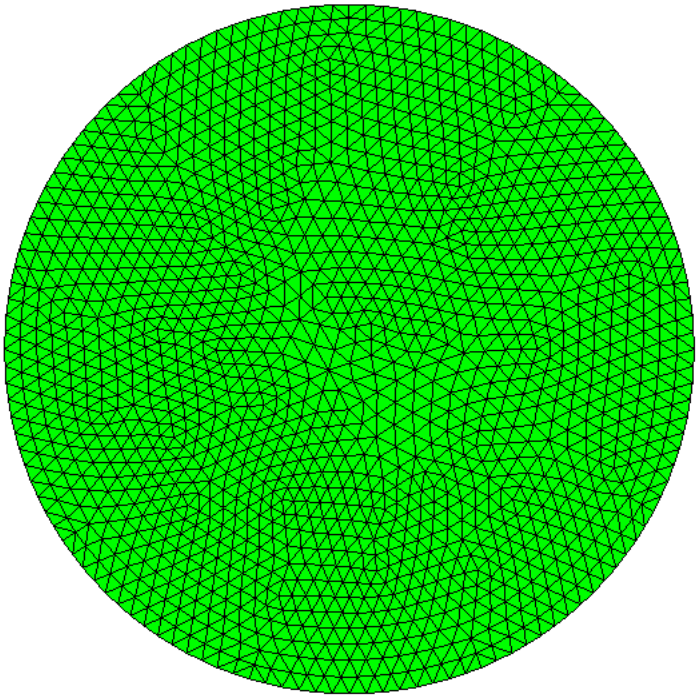


FIGURE 3.2: Circular geometry mesh with 2880 triangular elements

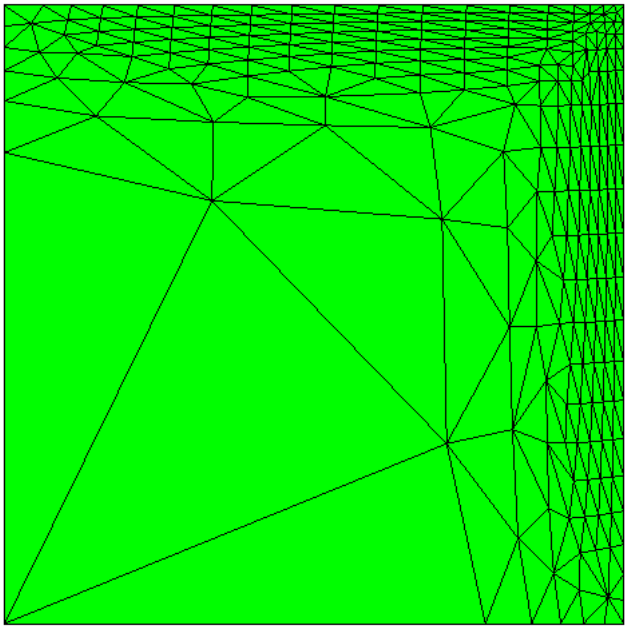


FIGURE 3.3: Mesh configuration 1 with 450 triangular elements

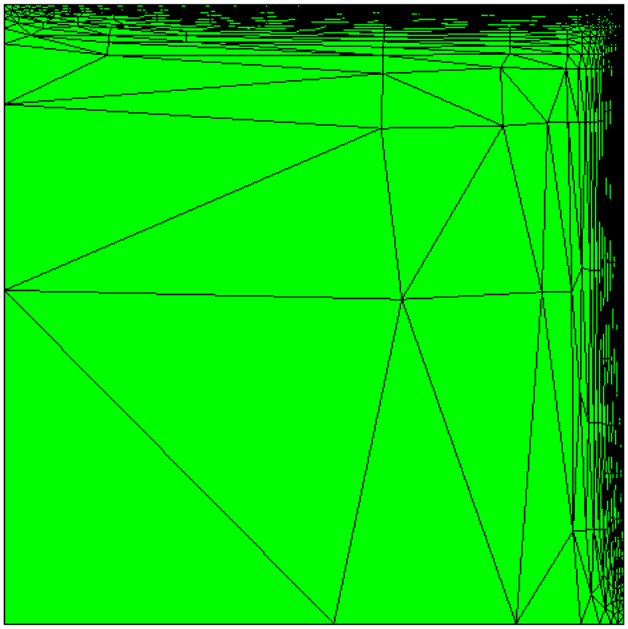


FIGURE 3.4: Mesh configuration 2 with 959 triangular elements

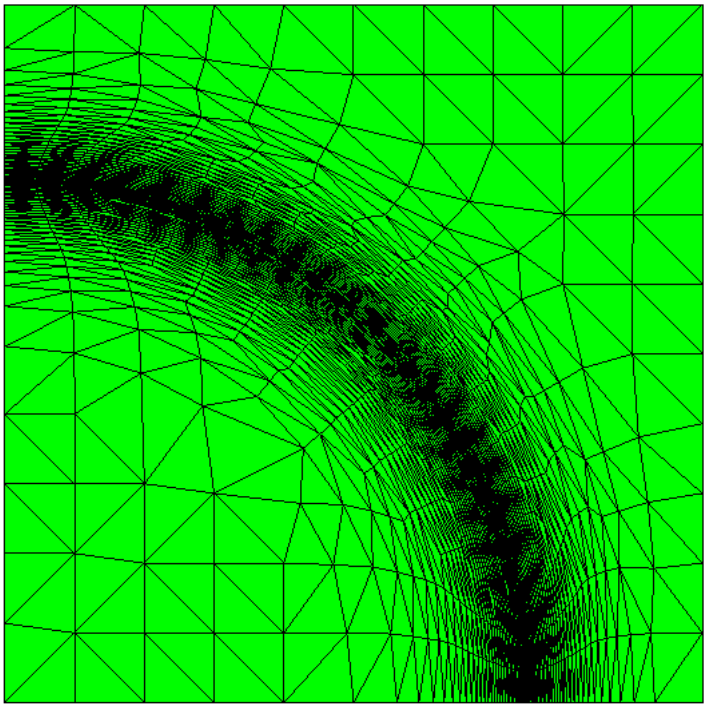
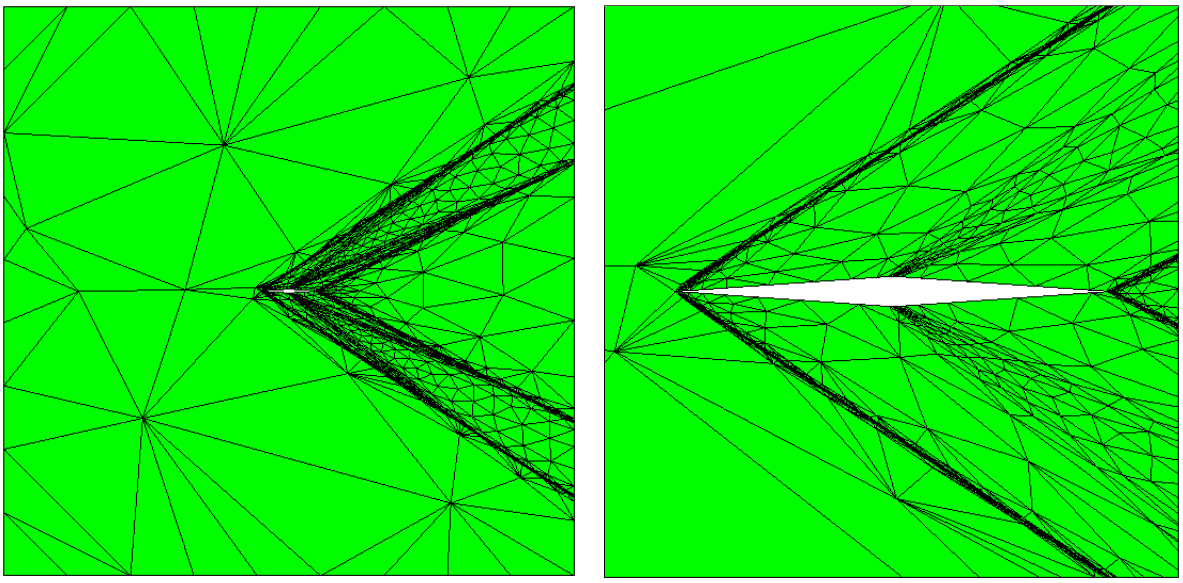


FIGURE 3.5: Mesh configuration 3 with 2264 triangular elements

3.2 Sorting methods

The elements in the mesh have to be assigned a priority order (the order in which they are to be recombined). There are various sorting techniques; some of the sorting techniques that are used in this thesis are mentioned below:



(A) The entire domain of the geometry. (B) Zoomed in on diamond airfoil.

FIGURE 3.6: Diamond airfoil mesh with 25038 triangular elements

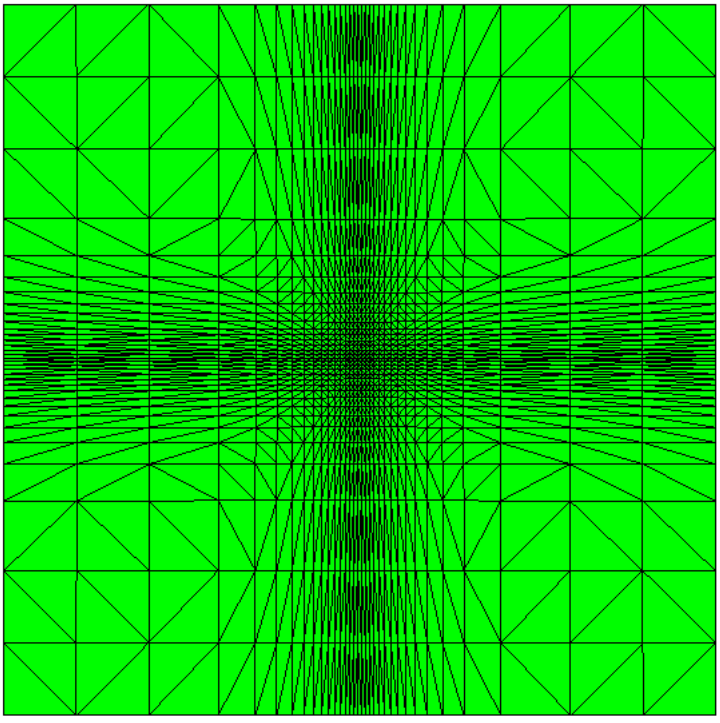


FIGURE 3.7: Cross geometry mesh with 2178 elements.

Note: ***"el_num"*** refers to the element number in the mesh, and ***"i"*** refers to the specific neighbor amongst the present neighbors to the current element.

In the figure 3.8 the naming conventions are as follows:

1. Triangle shaded in *green* $\Rightarrow$ Current element.

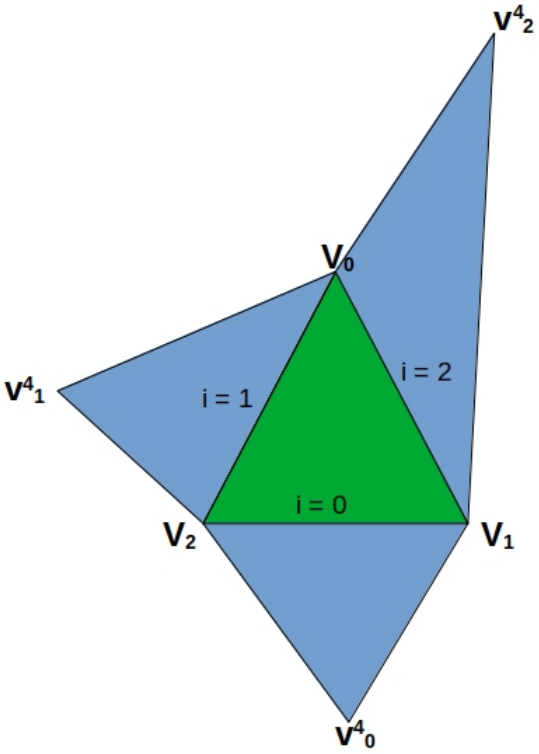


FIGURE 3.8: Convention used in the program

2. Triangles shaded in *blue* ⇒ neighboring element.

   (a) $\mathbf{V}_{i=0,1,2}$ ⇒ Vertices of the current element.

   (b) $\mathbf{v}^4_{i=0,1,2}$ ⇒ Vertices of the neighboring elements to the current element in consideration.

   (c) i = 0,1,2 ⇒ Corresponding neighbor numbers to the *local* current element.

3.2.1 Based on aspect ratios of elements

In this sorting method, the elements are organized in a decreasing fashion according to their aspect ratios, which implies that the highly skewed elements are assigned the highest priority for recombination.

The aspect ratio of a triangle is defined as the ratio of the longest edge to the shortest edge or the ratio of the circumradius to twice its inradius. The code snippet is given below: [1] [2].

[1] Note: The arguments to every function will be denoted by "el_num" and "i", these conventions are mentioned in the section **3.2** of chapter **3**.

[2] The "*vector_operations*" that are in the code snippets are a class of vector operations that are defined by the user (mentioned in chapter **4**) to speed the program up by bypassing the NumPy library.

```
def quality_func_aspect_ratio(el_num, i): # Calculates the aspect ratio of the element and its neighbor in consideration

    v = elements[el_num].vertices
    v0 = ma[v[0]].point
    v1 = ma[v[1]].point
    v2 = ma[v[2]].point

    if i == 0:
        v_opp = ma[neighbors_vert(el_num)[2][0]].point
        if v_opp is not None:
            sides = [vector_operations.vec_norm(np.subtract(v0, v1)), vector_operations.vec_norm(np.subtract(v1, v_opp)), vector_operations.vec_norm(np.subtract(v_opp, v2)), vector_operations.vec_norm(np.subtract(v2, v0))]
            aspect_ratio = max(sides)/min(sides)
        elif v_opp is None:
            aspect_ratio = 0

    if i == 1:
        v_opp = ma[neighbors_vert(el_num)[2][1]].point
        if v_opp is not None:
            sides = [vector_operations.vec_norm(np.subtract(v0, v1)), vector_operations.vec_norm(np.subtract(v1, v2)), vector_operations.vec_norm(np.subtract(v2, v_opp)), vector_operations.vec_norm(np.subtract(v_opp, v0))]
            aspect_ratio = max(sides)/min(sides)
        elif v_opp is None:
            aspect_ratio = 0

    if i == 2:
        v_opp = ma[neighbors_vert(el_num)[2][2]].point
        if v_opp is not None:
            sides = [vector_operations.vec_norm(np.subtract(v0, v_opp)), vector_operations.vec_norm(np.subtract(v1, v2)), vector_operations.vec_norm(np.subtract(v2, v0)), vector_operations.vec_norm(np.subtract(v_opp, v1))]
            aspect_ratio = max(sides)/min(sides)
        elif v_opp is None:
            aspect_ratio = 0

    return aspect_ratio
```

Listing 3.2: Function to calculate Aspect ratio

In the above code, the function is called whenever the aspect ratio of a certain quad is required (current element and its neighbor). For example:

```
for el in ma.Elements(VOL):
    # print(el.nr)
    # print(type(el.nr))        # int
    # print(type(el))          # <class 'ngsolve.comp.Ngs_Element'>
    # print(el)                  # <ngsolve.comp.Ngs_Element object at 0
    x7f8b3b3b3b70>

    v = el.vertices            # get the coordinates of the vertices of the
    element

    curr_el = el.nr
    neighbor_el = 0 # This is equivalent to i = 0; it can also be set to 1 or 2.

    # Function to calculate aspect ratios and store it in a variable called as "
    aspect_ratio".
    aspect_ratio_val = quality_func_aspect_ratio(curr_el, neighbor_el)
```

LISTING 3.3: Accessing the aspect ratio function

3.2.2 Based on an edge quality criterion

Each pair of adjacent triangles is likely to form a quadrilateral element and is identified by the shared edge. A simplified measure of the corresponding quadrilateral quality is associated with each edge. Let the green highlighted triangle be the current element and the neighbor selected in consideration be the element with neighbor number as 1; we end up with the following set of vectors: $[\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}]$ and $[\mathbf{v_2}, \mathbf{v_1^4}, \mathbf{v_0}]$ be two adjacent triangles, sharing the edge a = $[\mathbf{v_2}, \mathbf{v_0}]$. If $\alpha_0 = \left(v_1\vec{v}_0, v_1^4\vec{v}_0\right)$, $\alpha_1 = (v_2\vec{v}_1, v_0\vec{v}_1)$, $\alpha_2 = \left(v_1\vec{v}_2, v_1^4\vec{v}_2\right)$, $\alpha_3 = \left(v_2\vec{v}_1^4, v_0\vec{v}_1^4\right)$ then, the quality of edge a can be defined as (refer to the figure 3.9) [2, 7]:

$$q(a) = \mathbf{max}\left(1 - \frac{2}{\pi}\mathbf{max}\left(|\frac{\pi}{2} - \alpha_k|\right), 0\right) \tag{3.1}$$

Below is the code snippet where an argument with all internal angles is given, and it outputs the quality values. [3]

```
def quality_func_angles(internal_angles):

    if len(internal_angles) != 0:
```

[3] The argument - "*internal_angles*" to the function in the **code snippet listing 3.1** in section **3.2.2** of chapter **3** is explained further in chapter **4**.

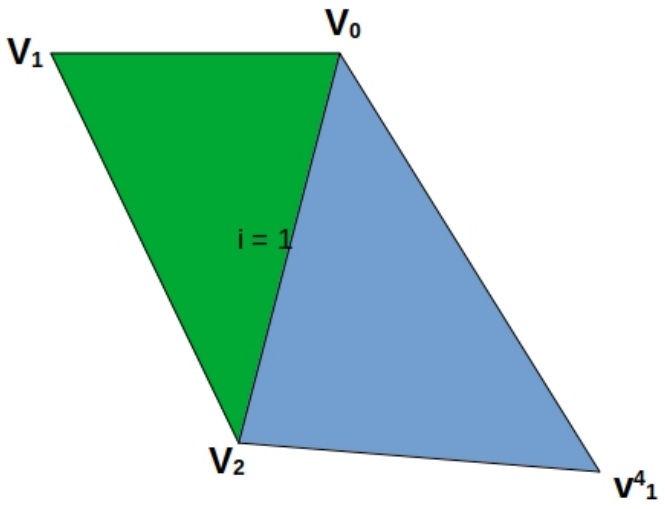


FIGURE 3.9: Edge quality depiction

```
4        quality_temp = max(abs(0.5*np.pi - internal_angles[0]), abs(0.5*np.pi -
    internal_angles[1]), abs(0.5*np.pi - internal_angles[2]), abs(0.5*np.pi -
    internal_angles[3]))
5        quality = max((1.0 - (2.0/np.pi)*quality_temp), 0)
6
7    elif len(internal_angles) == 0:
8        quality = -0.1 # Dummy negative value to indicate that the element is a
    boundary element so that it is also sorted without the "None" error
9
10   return quality
```

LISTING 3.4: Function which calculates the above mentioned quality from equation (3.1)

3.3 Linear combination of aspect ratio quality and edge quality

We pair the techniques mentioned in section **3.2.1** and section **3.2.2** to form a linear combination system. Both these quality values are then assigned a weight that can be changed accordingly. The equation below shows how both of them are combined into a linear combination: Let weight be denoted as ω;

$$q = \omega * \left(\frac{\text{aspect\_ratio\_local}}{\textbf{max}(\text{aspect\_ratio\_global})} \right) + (1 - \omega) * \left(\textbf{max} \left(1 - \frac{2}{\pi} \textbf{max} \left(|\frac{\pi}{2} - \alpha_k| \right), 0 \right) \right) \tag{3.2}$$

Chapter 4

Program setup for recombination

The libraries used in the program are:

1. **ngsolve** - imported Mesh and VOL from this library
2. **netgen**
3. **NumPy**
4. **math**

4.1 Element activity function

A function is created to turn elements **ON** or **OFF**. An element is realised that it is **ON** if the value of the **element_activity** corresponds to 1 and **OFF** if the value of the **element_activity** corresponds to 0.

The code snippet is shown below:

```
def activity():
    for el in ma.Elements(VOL):
        element_activity[el.nr] = 1
    return element_activity
```

Listing 4.1: Element activity

4.2 Element neighbors number

Function to find out the neighbors to a current element that is in consideration. This works by taking the intersection of the elements that share the vertices of the current element.

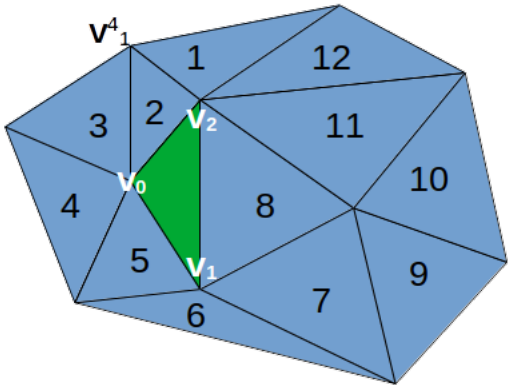


FIGURE 4.1: Depiction of working of neighbors_elm function

In the figure 4.1, let green be the current element having vertices $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$ and let the neighbors be numbered 1 through 12, for example, if we want to find $\mathbf{v}_1^4$ then we find the elements that share the vertices $\mathbf{v}_0$, $\mathbf{v}_2$ or the edge number 1.

Elements that share $\mathbf{v}_0 \implies$ (4, 3, 2, current element)
Elements that share $\mathbf{v}_2 \implies$ (2, 1, 12, current element)
So, intersection(4, 3, 2, current element, 2, 1, 12, current element) $\implies$ (2, current element)
Now, if the current element is removed from the above list, we get "2".
This implies that the element which is opposite to $\mathbf{v}_1$ and is neighbor to the current element is the element numbered "2".

```
1  def neighbors_elm(el_num):
2      v = elements[el_num].vertices
3
4      v0_list = []
5      v1_list = []
6      v2_list = []
7      # loop over all vertices
8      curr_el = elements[el_num].nr
9      for i in range(3):
10         for el in ma[v[i]].elements: # loop over all elements that contain the
       vertex v
11
12             if i == 0:
```

```
13                  v0_list.append(el.nr)
14
15              elif i == 1:
16                  v1_list.append(el.nr)
17
18              elif i == 2:
19                  v2_list.append(el.nr)
20
21      v4_0 = intersection(v1_list, v2_list)
22      v4_1 = intersection(v0_list, v2_list)
23      v4_2 = intersection(v0_list, v1_list)
24
25      v4_0.remove(curr_el)
26      v4_1.remove(curr_el)
27      v4_2.remove(curr_el)
28
29      if len(v4_0) < 1:
30          v4_0 = None
31      if len(v4_1) < 1:
32          v4_1 = None
33      if len(v4_2) < 1:
34          v4_2 = None
35
36      #print(j)
37      if v4_0 is not None:
38          v4_0 = int(*v4_0)
39      if v4_1 is not None:
40          v4_1 = int(*v4_1)
41      if v4_2 is not None:
42          v4_2 = int(*v4_2)
43
44      #gives the opposite element as per our convention
45      el_neighbors = [v4_0, v4_1, v4_2]
46
47      return el_neighbors
```

LISTING 4.2: Element neighbors numbers

4.3 Element neighbors vertices

This function evaluates three sources of information about the current element and its neighbors:

1. v_list ⇒ Coordinates of vertices of the current element.

2. v_list_ID ⇒ ID/number of the vertices of the current element.

3. v_listOpp ⇒ Coordinates of vertices of the opposite vertices (neighbors - $\mathbf{v}_0^4$, $\mathbf{v}_1^4$, $\mathbf{v}_2^4$)

The code snippet is shown below:

```
def neighbors_vert(el_num):

    for el in ma.Elements(VOL):
        el_neigbors = neighbors_elm(el.nr)
        v4_0List.append(el_neigbors[0])
        v4_1List.append(el_neigbors[1])
        v4_2List.append(el_neigbors[2])

    # Check if None is stored, if so print "None encountered"
    curr_el = elements[el_num].nr
    # print(i)
    if v4_0List[curr_el] is not None:
        k = v4_0List[curr_el]
        v4_0ID = [elm_vert_list[k][0], elm_vert_list[k][1], elm_vert_list[k][2]]
    else:
        v4_0ID = [None, None, None]
    # print(v4)
    if v4_1List[curr_el] is not None:
        k = v4_1List[curr_el]
        v4_1ID = [elm_vert_list[k][0], elm_vert_list[k][1], elm_vert_list[k][2]]
    else:
        v4_1ID = [None, None, None]
    #print(v4_1ID)

    if v4_2List[curr_el] is not None:
        k = v4_2List[curr_el]
        v4_2ID = [elm_vert_list[k][0], elm_vert_list[k][1], elm_vert_list[k][2]]
    else:
        v4_2ID = [None, None, None]

    el_vList = [elm_vert_list[curr_el][0], elm_vert_list[curr_el][1],
    elm_vert_list[curr_el][2]]
    el_vListArr[curr_el] = [elm_vert_list[curr_el][0].nr, elm_vert_list[curr_el
    ][1].nr, elm_vert_list[curr_el][2].nr]
    # print( v4_0ID, v4_1ID, v4_2ID, el_vList)
    if v4_0ID is not None :
        v4_0 = set(v4_0ID)^set(el_vList)
        v4_0.remove(el_vList[0])
    else:
        v4_0 = None

```

```
    if v4_1ID is not None:
        v4_1 = set(v4_1ID)^set(el_vList)
        v4_1.remove(el_vList[1])
    else:
        v4_1 = None

    if v4_2ID is not None:
        v4_2 = set(v4_2ID)^set(el_vList)
        v4_2.remove(el_vList[2])
    else:
        v4_2 = None

    v4_0 = list(v4_0)
    v4_1 = list(v4_1)
    v4_2 = list(v4_2)

    if len(v4_0) != 1:
            v4_0 = [None]
    if len(v4_1) != 1:
            v4_1 = [None]
    if len(v4_2) != 1:
            v4_2 = [None]

    v_list = ma[el_vList[0]].point, ma[el_vList[1]].point, ma[el_vList[2]].point
     #gives the vertex coordinates of the element in consideration
    v_list_ID = ma[el_vList[0]].nr, ma[el_vList[1]].nr, ma[el_vList[2]].nr #
    gives the vertex ID of the element in consideration
    v_listOpp = *v4_0, *v4_1, *v4_2 #gives opposite vertex (neighbor) to each
    vertex of the element in consideration

    return v_list, v_list_ID, v_listOpp
```

LISTING 4.3: Function to evaluate the coordinates of the neighbor vertices and the current element in consideration.

4.4 Vector operations class

A class performs vector operations in the program, replacing NumPy for speed-ups. This happens because the inherent NumPy function overhead is avoided. This operation is only efficient when using arrays or lists of small lengths, which is the case in this program.

The code snippet is shown below:

```
class vector_operations:
```

```
2
3     def vec_norm(vec):
4         vec_norm = sqrt(vec[0]**2 + vec[1]**2)
5         return vec_norm
6
7     def dot_product(vec1, vec2):
8         dot = vec1[0]*vec2[0] + vec1[1]*vec2[1]
9         return dot
10
11    def vec_subtract(vec1, vec2):
12        vec_sub = [vec1[0]-vec2[0], vec1[1]-vec2[1]]
13        return vec_sub
```

Listing 4.4: Class that performs vector operations.

4.5 Internal angles

The function below evaluates all the internal angles, and then it is used as an argument to other functions like the quality function associated with edges.
The code snippet is shown below:

```
1  def internal_angles(el_num, i):
2
3      neighbor_vertices = neighbors_vert(el_num)[2]
4      element_vertices = neighbors_vert(el_num)[0]
5      neighbor_vertices_coords = []
6      internal_angles = []
7
8      for j in range(3):
9          if neighbor_vertices[j] is not None:
10             neighbor_vertices_coords.append(ma[neighbor_vertices[j]].point)
11         elif neighbor_vertices[j] is None:
12             neighbor_vertices_coords.append(None)
13
14     if i == 0:
15         quad_coords = element_vertices[0], element_vertices[1],
       neighbor_vertices_coords[0], element_vertices[2]
16         if quad_coords[2] is not None:
17             for vertex in range(4):
18                 # vec_1 = tuple(np.subtract(quad_coords[(vertex+1)%4],
       quad_coords[vertex]))
19                 # vec_2 = tuple(np.subtract(quad_coords[(vertex+3)%4],
       quad_coords[vertex]))
```

```
20                vec_1 = vector_operations.vec_subtract(quad_coords[(vertex+1)
    %4], quad_coords[vertex])
21                vec_2 = vector_operations.vec_subtract(quad_coords[(vertex+3)
    %4], quad_coords[vertex])
22                # print(vec_1, vec_2)
23                dot = vector_operations.dot_product(vec_1, vec_2)
24                mag_vec_1 = vector_operations.vec_norm(vec_1)
25                mag_vec_2 = vector_operations.vec_norm(vec_2)
26                cos_theta = dot/(mag_vec_1*mag_vec_2)
27                theta_rad = np.arccos(np.clip(cos_theta, -1.0, 1.0))
28                # theta_deg = np.degrees(theta_rad)
29
30                internal_angles.append(theta_rad)
31
32            elif neighbor_vertices_coords[0] is None:
33                internal_angles.append(float('nan'))
34
35
36        if i == 1:
37            quad_coords = element_vertices[0], element_vertices[1], element_vertices
    [2], neighbor_vertices_coords[1]
38            if quad_coords[3] is not None:
39                for vertex in range(4):
40                    # vec_1 = tuple(np.subtract(quad_coords[(vertex+3)%4],
    quad_coords[vertex]))
41                    # vec_2 = tuple(np.subtract(quad_coords[(vertex+1)%4],
    quad_coords[vertex]))
42                    vec_1 = vector_operations.vec_subtract(quad_coords[(vertex+3)
    %4], quad_coords[vertex])
43                    vec_2 = vector_operations.vec_subtract(quad_coords[(vertex+1)
    %4], quad_coords[vertex])
44                    # print(vec_1, vec_2)
45                    dot = vector_operations.dot_product(vec_1, vec_2)
46                    mag_vec_1 = vector_operations.vec_norm(vec_1)
47                    mag_vec_2 = vector_operations.vec_norm(vec_2)
48                    cos_theta = dot/(mag_vec_1*mag_vec_2)
49                    theta_rad = np.arccos(np.clip(cos_theta, -1.0, 1.0))
50                    # theta_deg = np.degrees(theta_rad)
51
52                    internal_angles.append(theta_rad)
53
54            elif neighbor_vertices_coords[0] is None:
55                internal_angles.append(float('nan'))
56
57        if i == 2:
58            quad_coords = element_vertices[0], neighbor_vertices_coords[2],
    element_vertices[1], element_vertices[2]
59            if quad_coords[1] is not None:
```

```
60            for vertex in range(4):
61                # vec_1 = tuple(np.subtract(quad_coords[(vertex+1)%4],
    quad_coords[vertex]))
62                # vec_2 = tuple(np.subtract(quad_coords[(vertex+3)%4],
    quad_coords[vertex]))
63                vec_1 = vector_operations.vec_subtract(quad_coords[(vertex+1)
    %4], quad_coords[vertex])
64                vec_2 = vector_operations.vec_subtract(quad_coords[(vertex+3)
    %4], quad_coords[vertex])
65                # print(vec_1, vec_2)
66                dot = vector_operations.dot_product(vec_1, vec_2)
67                mag_vec_1 = vector_operations.vec_norm(vec_1)
68                mag_vec_2 = vector_operations.vec_norm(vec_2)
69                cos_theta = dot/(mag_vec_1*mag_vec_2)
70                theta_rad = np.arccos(np.clip(cos_theta, -1.0, 1.0))
71                # theta_deg = np.degrees(theta_rad)
72
73                internal_angles.append(theta_rad)
74
75        elif neighbor_vertices_coords[0] is None:
76            internal_angles.append(float('nan'))
77
78
79    return internal_angles
```

LISTING 4.5: Function that returns the internal angles of a given quad (current element + neighbor).

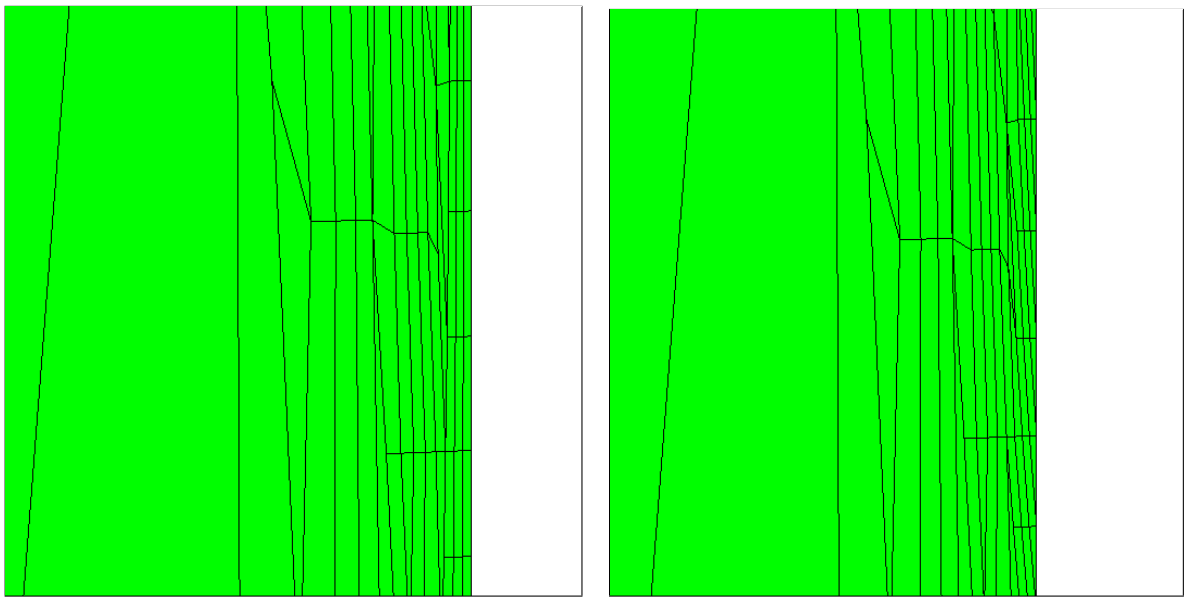
4.6 Boundary weights

This function assigns the boundary element some weight by multiplying the associated quality function by 1.5. Boundary weights prioritize boundary elements during recombination, improving accuracy in areas critical to numerical simulations, such as near boundaries or interfaces. Boundary elements, often critical for accurate computations, are prioritized by weighting their quality function, as illustrated in the figure 4.2.

The figure 4.2(A) shows a recombined mesh with boundary weights, forming 512 elements, while 4.2(B) shows the recombined mesh with 527 elements without weights, the elements in 4.2(A) near boundary wall show a gradual transition and more quads at the boundaries as compared to in the 4.2(B) highlighting the importance of this weighting function. This function can also be turned off or avoided if the boundary weighting is not necessary. The code snippet is shown below:

```
def boundary_weights(el_num):

    # for el in ma.Elements(BND):
    #     if elm_info[0] == el.nr:
    #         elm_info[4] *= 1.5

    el_neighbors = neighbors_elm(el_num)

    if el_neighbors[0] is None or el_neighbors[1] is None or el_neighbors[2] is
    None:
        for bnd in range(len(elm_info)):
            if elm_info[bnd][0] == el_num:
                elm_info[bnd][4] *= 1.5

    return elm_info
```

Listing 4.6: Function that associates the boundary elements with some weight.



(a) The boundary elements form a quad when the weights are applied. 512 elements

(b) Boundary elements remain a triangular element without weights. 527 elements

Figure 4.2: Difference between weights and no weights.

4.7 Main body

```
# read mesh file
ma = Mesh("mesh.vol") # Write your mesh filename inside the " "
n_vert = ma.nv
n_el = ma.ne
n_edge = ma.nedge
```

```
d = np.sqrt(1)  # length of each edge in metric space
d = d**2

metric_loc = np.zeros((n_el, 3))  # store the metric tensor for each element
edge_qual_heap_temp = []
edge_qual_heap_temp2 = []
edge_qual = [[None for _ in range(3)] for _ in range(n_el)]
edge_qual_temp = []
edge_qual_heap = []
edge_qual_tempLst = [[None for _ in range(3)] for _ in range(n_el)]
edge_qual_heapSorted = []
element_activity = np.zeros((n_el,1))

edge_activity_temp = np.zeros((n_el,3))
edge_activity = np.zeros((n_el*3,1))

aspect_ratioSorted_el = np.zeros((n_el, 1))
quad_listArr = []

v4_0List = []
v4_1List = []
v4_2List = []
v_listOpp = []
v_list = []
v_list_ID = []

el_vListArr = np.zeros((n_el, 3))
v_oppListArr = np.zeros((n_el,3))

opp_vertices = [[0,0], [0,0], [0,0]]
opp_vertices_list = [[0] * 3 for i in range(n_el)]
# opp_vertices_list = ()
aspectRatioThreshold = 1 # threshold for aspect ratio

elements = []
for el in ma.Elements(VOL):
    elements.append(el)
    # print(el)

# loop over all elements
for el in ma.Elements(VOL):
    # print(el.nr)
    # print(type(el.nr))      # int
    # print(type(el))         # <class 'ngsolve.comp.Ngs_Element'>
    # print(el)               # <ngsolve.comp.Ngs_Element object at 0
    x7f8b3b3b3b70>

```

```
52      v = el.vertices           # get the coordinates of the vertices of the
        element
53      # print(el.nr,v)
54      # get coordinates of v
55      v0 = ma[v[0]].point
56      v1 = ma[v[1]].point
57      v2 = ma[v[2]].point
58
59      alpha = np.array([v1[0] - v0[0], v2[0] - v1[0], v0[0] - v2[0]])
60      beta = np.array([v1[1] - v0[1], v2[1] - v1[1], v0[1] - v2[1]])
61
62      # coefficients of system of 3 linear equations
63      a = np.array([alpha[0]**2, alpha[1]**2, alpha[2]**2])
64      c = np.array([beta[0]**2, beta[1]**2, beta[2]**2])
65      b = np.array([2*alpha[0]*beta[0], 2*alpha[1]*beta[1], 2*alpha[2]*beta[2]])
66
67      # determinant of 3*3 linear system
68      det = a[0]*(b[1]*c[2] - b[2]*c[1]) - b[0]*(a[1]*c[2] - a[2]*c[1]) + c[0]*(a
        [1]*b[2] - a[2]*b[1])
69
70      detx = d*(b[1]*c[2] - b[2]*c[1]) - b[0]*(d*c[2] - d*c[1]) + c[0]*(d*b[2] - d
        *b[1])
71      dety = a[0]*(d*c[2] - d*c[1]) - d*(a[1]*c[2] - a[2]*c[1]) + c[0]*(a[1]*d - a
        [2]*d)
72      detz = a[0]*(b[1]*d - b[2]*d) - b[0]*(a[1]*d - a[2]*d) + d*(a[1]*b[2] - a
        [2]*b[1])
73
74      # value of unknowns (x , y, z) which are also 3 independent entries of the
        2*2 metric tensor [x, y; y, z]
75      x = detx/det
76      y = dety/det
77      z = detz/det
78      metric_loc[el.nr] = [x, y, z]    # el.nr gives the element number
79
80  elm_vert_list = np.zeros((n_el,3), dtype = object)   # list
81  for el in ma.Elements(VOL):
82      v = el.vertices
83      elm_vert_list[el.nr] = v
84
85  edges_list = np.zeros((n_edge,3), dtype = object)
86  for edges in ma.edges:
87      edges_list[edges.nr] = edges
88
89  # Finding the metric tensor at each node by doing the average of the metric
        tensor of the elements that contain the node
90  implied_metric = np.zeros((n_vert, 3)) # store the metric tensor for each vertex
91
92  for v in ma.vertices:        # loop over all vertices
```

```
93      # print(v.nr)
94      vol = 0
95      for el in ma[v].elements: # loop over all elements that contain the vertex v
96
97          v0 = ma[elm_vert_list[el.nr][0]].point  # get the coordinates of element
        el
98          v1 = ma[elm_vert_list[el.nr][1]].point
99          v2 = ma[elm_vert_list[el.nr][2]].point
100
101         # calculate volume of el
102         vol_loc = 1/6*np.abs((v1[0] - v0[0])*(v2[1] - v0[1]) - (v2[0] - v0[0])*(
      v1[1] - v0[1]))
103
104         # add metric tensor of el to the metric tensor of v
105         implied_metric[v.nr] += vol_loc*metric_loc[el.nr]
106         vol += vol_loc
107
108     # divide by the volume of the element to get the volume average of metric at
        node v
109     implied_metric[v.nr] /= vol
110     implied_metric[v.nr] = implied_metric[v.nr]
111
112     # scale down the implied_metric
113     # implied_metric[v.nr] = implied_metric/4
114
115     # with open ('implied_metric.txt', 'a') as filehandle:
116     #     print(implied_metric[v.nr], v.nr, file=filehandle)
117
118 activity()
119
120 # for i in range(len(edges_list)):
121 #     print(edges_list[i].nr)
122
123 print(f"Element sorting based on quality function has started")
124
125 aspect_ratio_max = 0
126
127 weight = 0.8
128 start_time = time.time()
129 for el in ma.Elements(VOL):
130
131     neighbor = neighbors_elm(el.nr)
132     aspect_ratio_local = np.zeros((3,1))
133
134     for i in range(3):
135
136         if neighbor[i] is not None:
```

```
137             aspect_ratio = quality_func_aspect_ratio(el.nr, i) # temporarily
    storing the aspect ratio quality function
138             # quality_func = 2
139             aspect_ratio_local[i] = aspect_ratio
140 
141         else:
142             aspect_ratio = 0
143             aspect_ratio_local[i] = aspect_ratio
144 
145     aspect_ratio_next_el = max(aspect_ratio_local)
146     if aspect_ratio_next_el > aspect_ratio_max:
147         aspect_ratio_max = aspect_ratio_next_el
148 
149     print(f"---- Populating aspect ratio list for normalization ---- {(el.nr/
    n_el)*100} % completed", end= '\r')
150 
151 end_time_aspect_ratio = time.time()
152 print(f"Populating aspect ratio list for normalization has finished in {
    end_time_aspect_ratio-start_time} seconds")
153 
154 max_aspect_ratio = aspect_ratio_max
155 
156 # print(max_aspect_ratio)
157 
158 # elm_info = np.zeros((n_el*3, 6))
159 elm_info = []
160 
161 for el in ma.Elements(VOL):
162 
163     neighbor = neighbors_elm(el.nr)
164 
165     for i in range(3):
166 
167         if neighbor[i] is not None:
168             edge_vertex = intersection(elements[neighbor[i]].vertices, elements[
    el.nr].vertices)
169             common_edge = intersection(ma[edge_vertex[0]].edges, ma[edge_vertex
    [1]].edges)
170             quality_func = weight*(quality_func_aspect_ratio(el.nr, i)/
    max_aspect_ratio) + (1-weight)*quality_func_angles(internal_angles(el.nr, i)
    ) # ADJUSTING WEIGHTS HERE
171             elm_info.append([el.nr, neighbor[i], i, edges_list[common_edge[0].nr
    ][0], quality_func, 1]) # Changing this also would require changes below in
    the recombination part
172 
173         elif neighbor[i] is None:
174             edge_vertex = None # This "None" implies that this is a boundary
    edge
```

```
175             common_edge = None # This "None" implies that this is a boundary
    edge therefore common_edge is not required
176             # elm_info.append([el.nr, neighbor[i], i, None, quality_func_angles(
    internal_angles(el.nr, i)), 0])
177 
178     print(f"---- Element {el.nr} has been processed ---- {(el.nr/n_el)*100} %
    completed", end= '\r')
179 
180 end_time_elm_info = time.time()
181 print(f"Time taken for elm_info computation: {end_time_elm_info -
    end_time_aspect_ratio} seconds")
182 
183 for el in ma.Elements(VOL):
184     boundary_weights(el.nr)
185 
186 print(f"Boundary weights have been applied")
187 
188 elm_info.sort(key = lambda x: x[4], reverse = True)
189 
190 # elm_info_for_edge = copy.deepcopy(elm_info)
191 # elm_info_for_edge.sort(key = lambda x: x[0], reverse = False)
192 
193 
194 print(f"Element sorting based on quality function has finished")
195 
196 with open('quality_check_serial.txt', 'a') as filehandle:
197     for i in range(len(elm_info)):
198         print(elm_info[i], file=filehandle)
199 
200 iterations = 0
201 iterations_island_triangles = 0
202 for recombine in enumerate(elm_info):
203 
204     curr_el_nr = recombine[1][0]
205     edge_activity = recombine[1][5]
206     curr_el_neighbor_nr = recombine[1][1]
207     elm_edge_nr = recombine[1][2]
208     edge_info = recombine[1][3]
209 
210     v0_nr = neighbors_vert(curr_el_nr)[1][0]
211     v1_nr = neighbors_vert(curr_el_nr)[1][1]
212     v2_nr = neighbors_vert(curr_el_nr)[1][2]
213 
214     if element_activity[curr_el_nr] == 1 and edge_activity == 1:
215 
216         with open('recombined.txt', 'a') as filehandle:
217 
218             if elm_edge_nr == 0:
```

```
219                 if element_activity[curr_el_neighbor_nr] == 1:
220                     print("2"," ","1", " ", "0", " ", "0", "4", v0_nr+1, v1_nr
    +1, neighbors_vert(curr_el_nr)[2][0].nr+1, v2_nr+1, file=filehandle)
221
222                     element_activity[curr_el_nr] = 0
223                     element_activity[curr_el_neighbor_nr] = 0
224
225                     edges_curr_el = elements[curr_el_nr].edges
226                     edges_curr_el_neighbor = elements[curr_el_neighbor_nr].edges
227
228             if elm_edge_nr == 1:
229                 if element_activity[curr_el_neighbor_nr] == 1:
230                     print("2"," ","1", " ", "0", " ", "0", "4", v0_nr+1, v1_nr
    +1, v2_nr+1, neighbors_vert(curr_el_nr)[2][1].nr+1, file=filehandle)
231
232                     element_activity[curr_el_nr] = 0
233                     element_activity[curr_el_neighbor_nr] = 0
234
235                     edges_curr_el = elements[curr_el_nr].edges
236                     edges_curr_el_neighbor = elements[curr_el_neighbor_nr].edges
237
238             if elm_edge_nr == 2:
239                 if element_activity[curr_el_neighbor_nr] == 1:
240                     print("2"," ","1", " ", "0", " ", "0", "4", v0_nr+1,
    neighbors_vert(curr_el_nr)[2][2].nr+1, v1_nr+1, v2_nr+1, file=filehandle)
241
242                     element_activity[curr_el_nr] = 0
243                     element_activity[curr_el_neighbor_nr] = 0
244
245                     edges_curr_el = elements[curr_el_nr].edges
246                     edges_curr_el_neighbor = elements[curr_el_neighbor_nr].edges
247
248
249         percentage = ((2*(iterations+1))/n_el)*100
250         print(f"{iterations} : Element {curr_el_nr} and Element {
    curr_el_neighbor_nr} recombined ---- {percentage}% completed")
251         iterations += 1
252
253         elms_recombined = iterations
254
255
256 isolated_triangles = 0
257 iterations_island_triangles = iterations
258 for el in ma.Elements(VOL):
259
260     if element_activity[el.nr] == 1:
261
262         isolated_triangles += 1
```

```
263
264         v0_nr = neighbors_vert(el.nr)[1][0]
265         v1_nr = neighbors_vert(el.nr)[1][1]
266         v2_nr = neighbors_vert(el.nr)[1][2]
267
268         with open('recombined.txt', 'a') as filehandle:
269             print("2"," ","1", " ", "0", " ", "0", "3", v0_nr+1, v1_nr+1, v2_nr
    +1, file=filehandle)
270
271         percentage = ((elms_recombined*2 + isolated_triangles)/n_el)*100
272         print(f"{iterations_island_triangles} : Element {el.nr} is an island
    triangle ---- {percentage}% completed")
273         iterations_island_triangles += 1
```

LISTING 4.7: Function that associates the boundary elements with some weight.

The flowchart summary for the program is shown in figure 4.4. The program starts by evaluating metric field values at each node. Then, it calculates the maximum aspect ratio to normalize all the values used later in populating elm_info using the quality functions (refer to equation 3.2 in chapter **3**). The algorithm used to calculate the maximum aspect ratio is shown in figure 4.3.

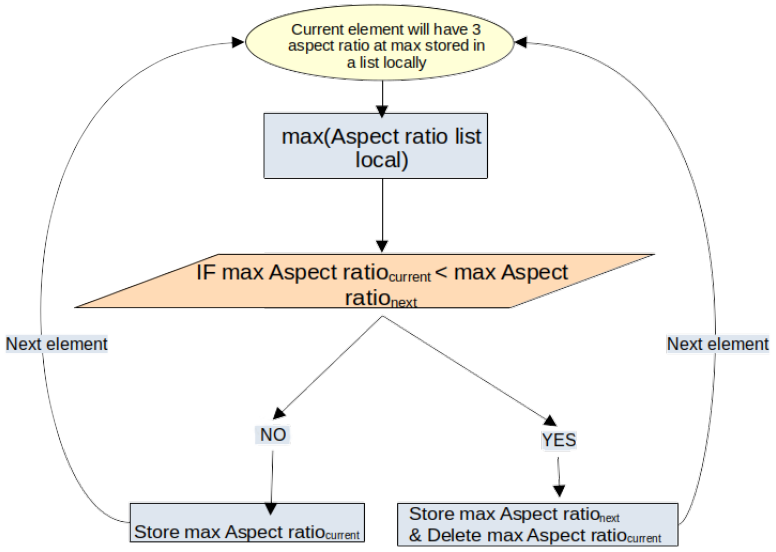


FIGURE 4.3: Flowchart to evaluate maximum aspect ratio algorithm

4.8 Bash script for automation

A bash script is executed to automate the program and output the "**.vol**" file directly in a folder. The script is listed below:

```
#!/bin/bash

rm -rf vol_files
mkdir vol_files
touch mesh_quad_original.vol
cp adap2-aflr-536.vol mesh_quad_original.vol # Change the first name of the file
     accordingly to your mesh file

for k in 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 # looping over different
    weights
do

    rm -f quad_max${k}.txt

    # The numbers 411 and 460,538 are the line numbers in the Python program
    where weight and file names are extracted from. If the code is changed,
    change these numbers and the Python program name accordingly. Here, it is
    quad_max.py
    sed -i "402s/weight = .*/weight = ${k}/g" quad_max.py
    sed -i "466s/with open('quad_max.*.txt', 'a') as filehandle:/with open('
    quad_max${k}.txt', 'a') as filehandle:/g" quad_max.py
    sed -i "544s/with open('quad_max.*.txt', 'a') as filehandle:/with open('
    quad_max${k}.txt', 'a') as filehandle:/g" quad_max.py

    echo "Running for weight = ${k}"

    python quad_max.py

    if [ $? -eq 0 ]; then
    echo "Success"
    else
    echo "Failed"
    fi

    # a=$(grep "surfaceelements" "mesh_quad_original.vol" | cut -d: -f1)
    start_line=$(cat -n mesh_quad_original.vol | grep "surfaceelements" | awk '{
    print $1}') #start line
    end_line=$(cat -n mesh_quad_original.vol | grep "volumeelements" | awk '{
    print $1}') #end line

    # grabbing the start and end lines in the file to copy
    c=$(expr $start_line)
```

```
34      d=$(expr $end_line - 2)
35
36      total_elements=$(wc -l quad_max${k}.txt | awk '{print $1}')
37
38      touch mesh_quad_${k}.vol
39      cp mesh_quad_original.vol mesh_quad_${k}.vol
40
41      head -n $((c)) "mesh_quad_${k}.vol" > temp_file.txt
42
43      echo $((total_elements)) >> temp_file.txt
44
45      cat "quad_max${k}.txt" >> temp_file.txt
46      tail -n +$((d)) "mesh_quad_${k}.vol" >> temp_file.txt
47      mv temp_file.txt "mesh_quad_${k}.vol"
48
49      mv mesh_quad_${k}.vol vol_files/ # moving the files to a new directory
50
51      rm -f quad_max${k}.txt # deleting the temporary files
52
53  done
54
55  rm -f mesh_quad_original.vol
```

Listing 4.8: Bash script for automation.

In the bash script in the listing 4.8, the loop is iterated for different weight values by the index "k", which can be changed according to the requirements.

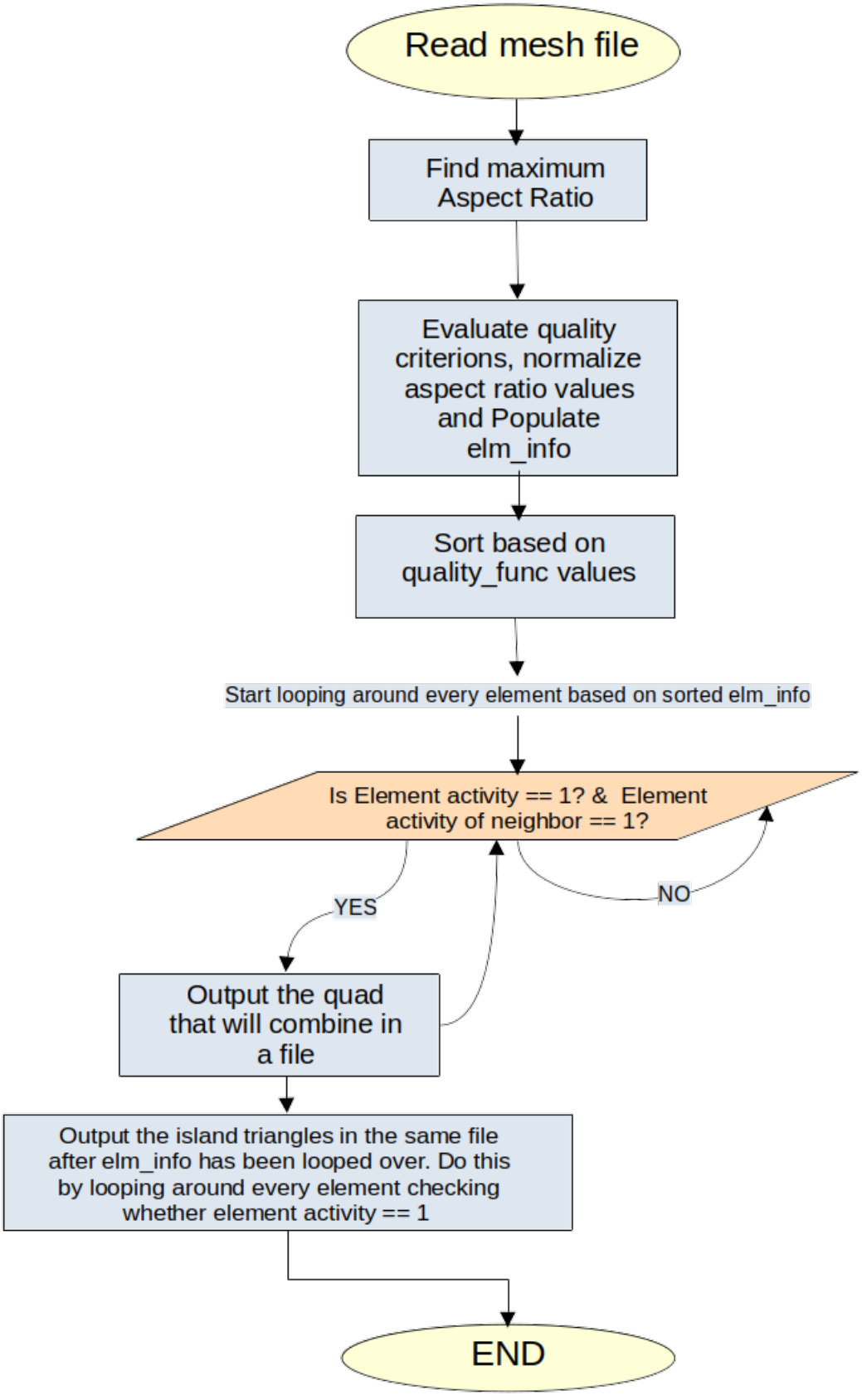


FIGURE 4.4: Flowchart of the program

Chapter 5

Results

5.1 Quarter circular geometry mesh

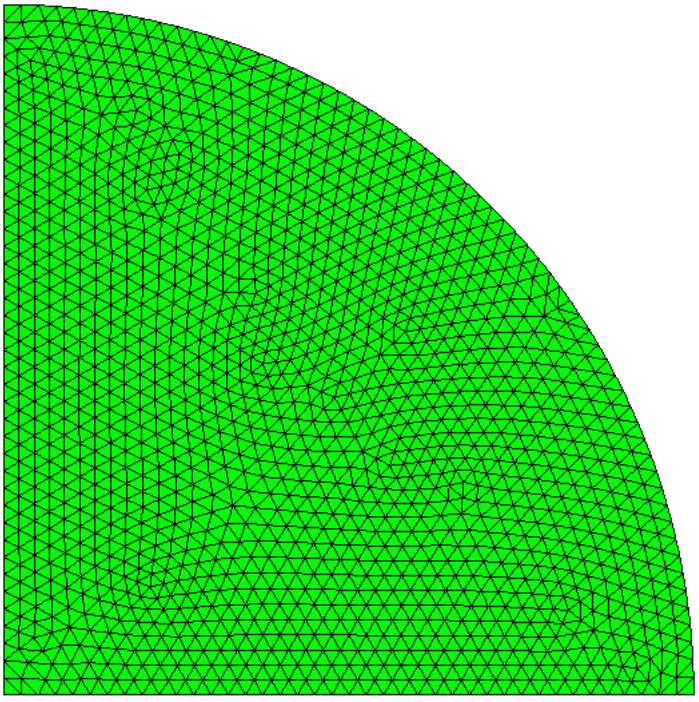


FIGURE 5.1: Quarter circle initial mesh

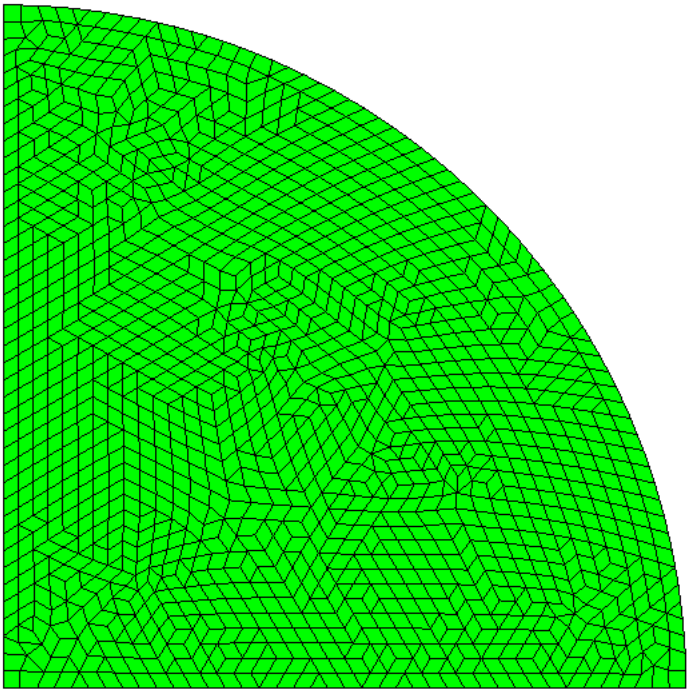


FIGURE 5.2: Quarter circle recombined mesh

5.2 Circular geometry mesh

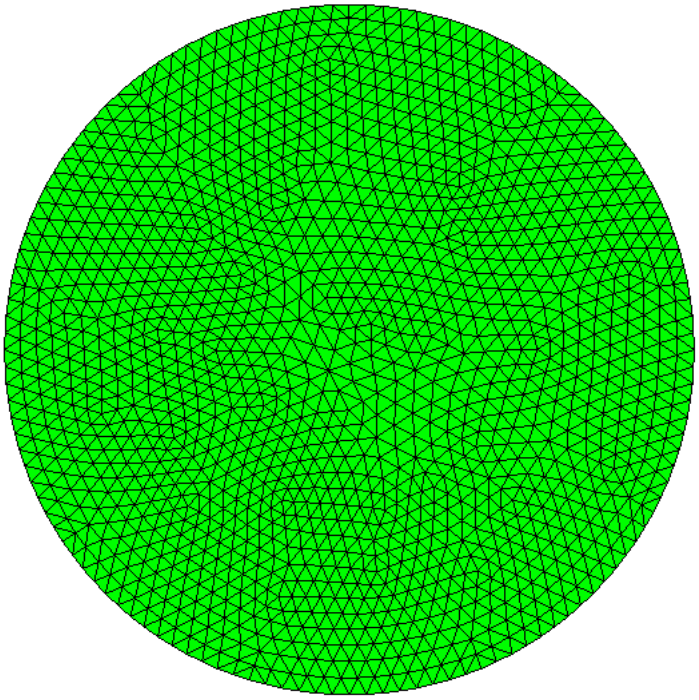


FIGURE 5.3: Circular initial mesh

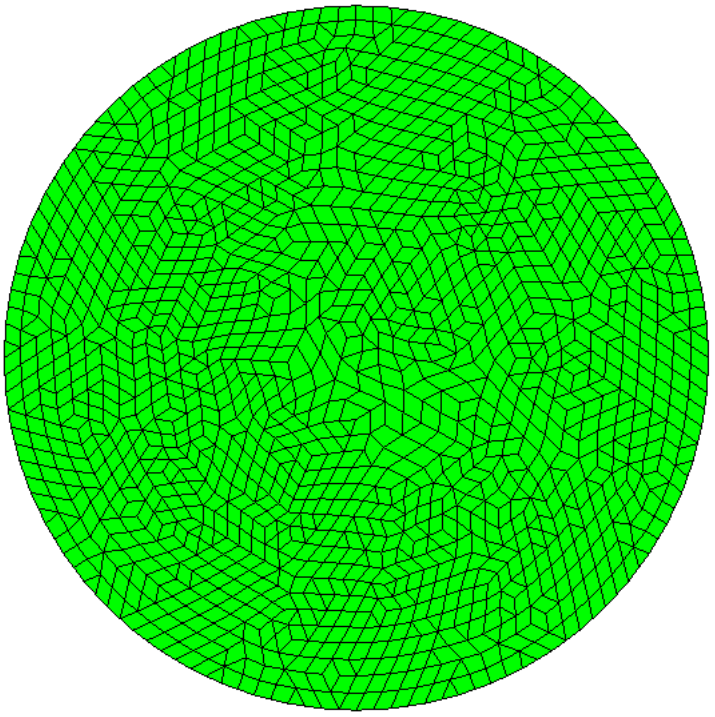


FIGURE 5.4: Circular recombined mesh

5.3 Mesh configuration 1

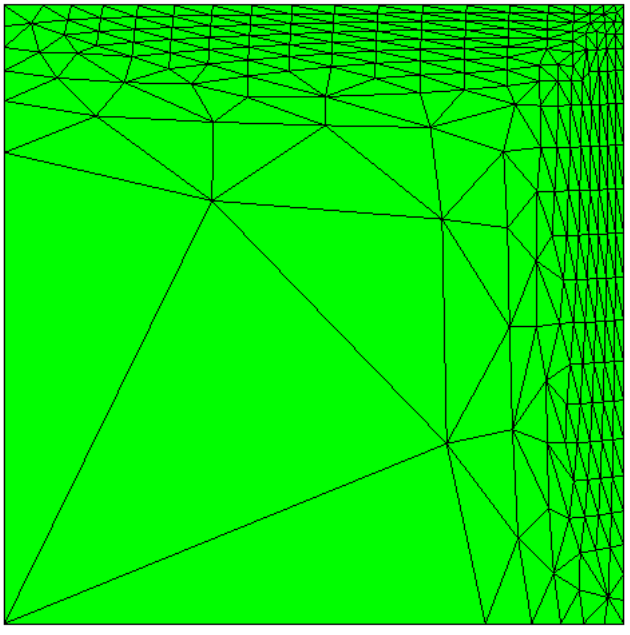


FIGURE 5.5: Mesh configuration 1 initial mesh

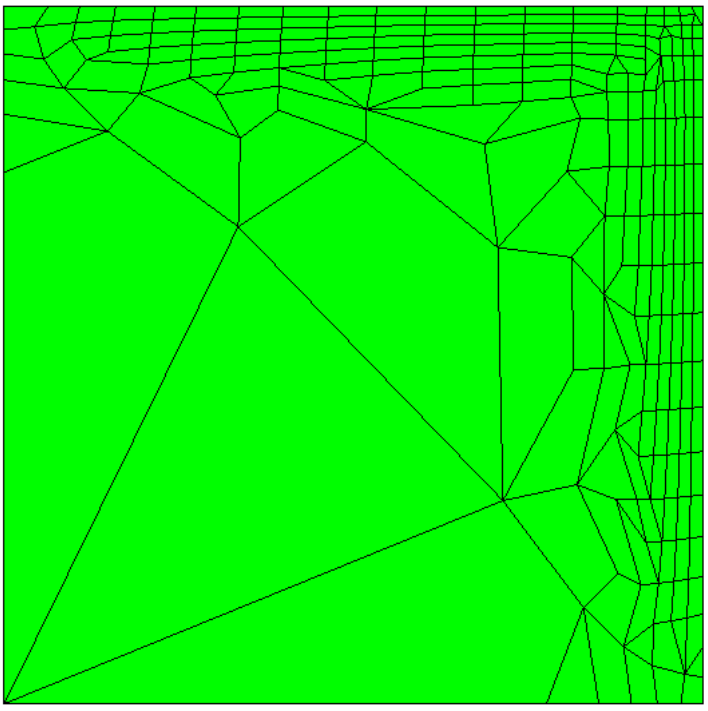


FIGURE 5.6: Mesh configuration 1 recombined mesh

5.4 Mesh configuration 2

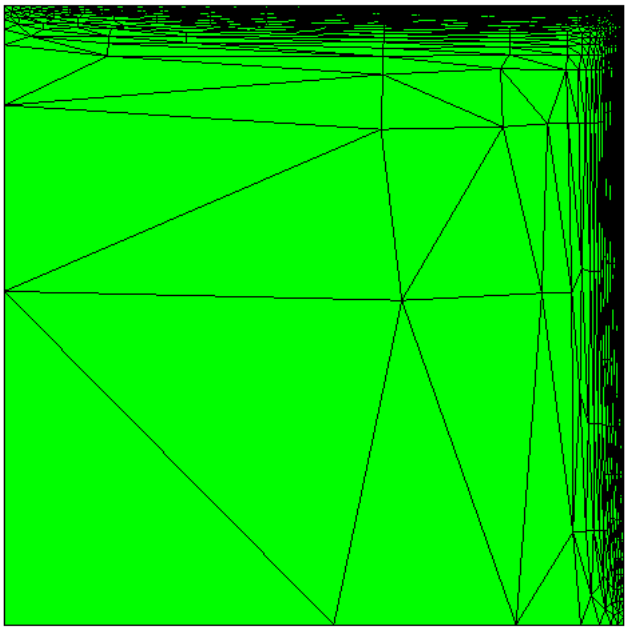
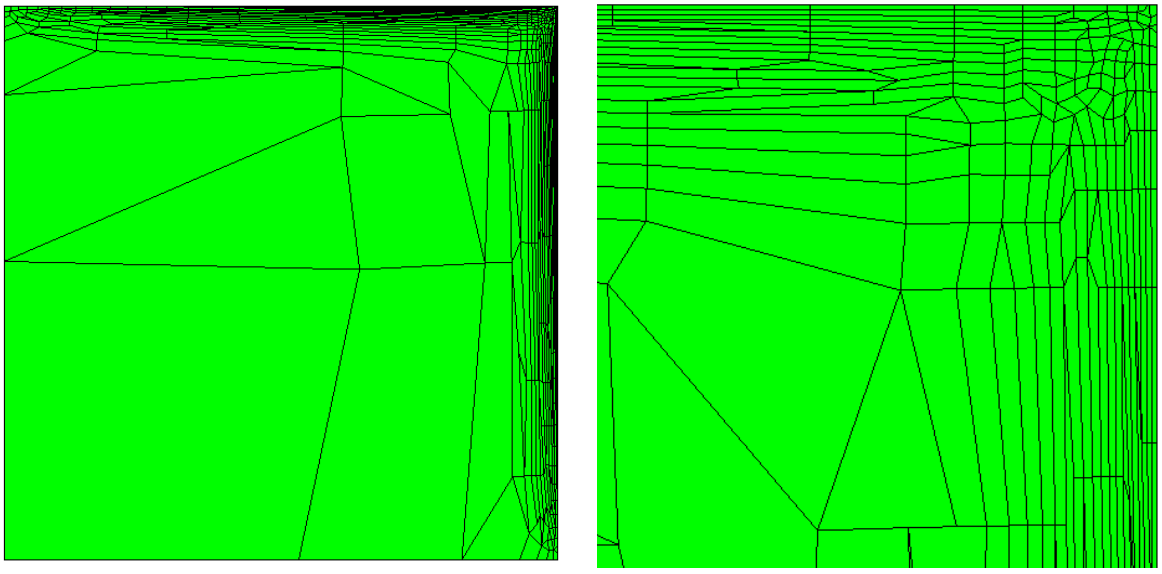


FIGURE 5.7: Mesh configuration 2 initial mesh



(A) Mesh zoomed out, full domain.

(B) Mesh zoomed up near the boundary.

FIGURE 5.8: Mesh configuration 2 recombined mesh.

5.5 Mesh configuration 3

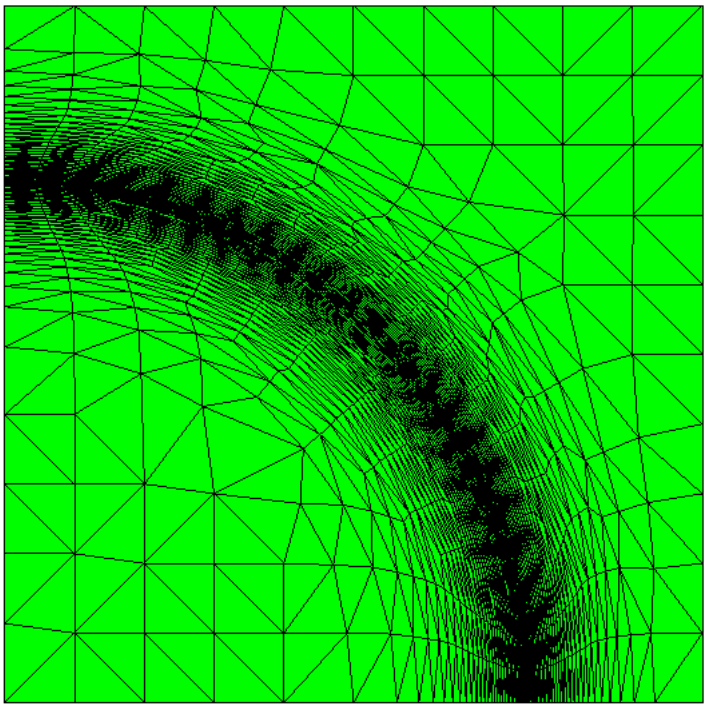
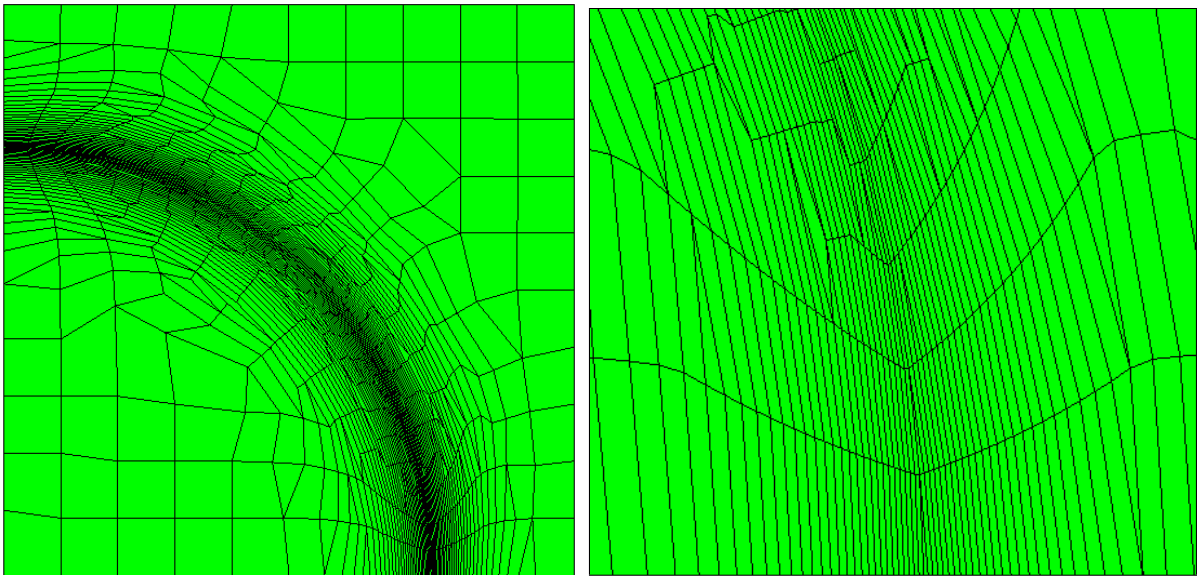


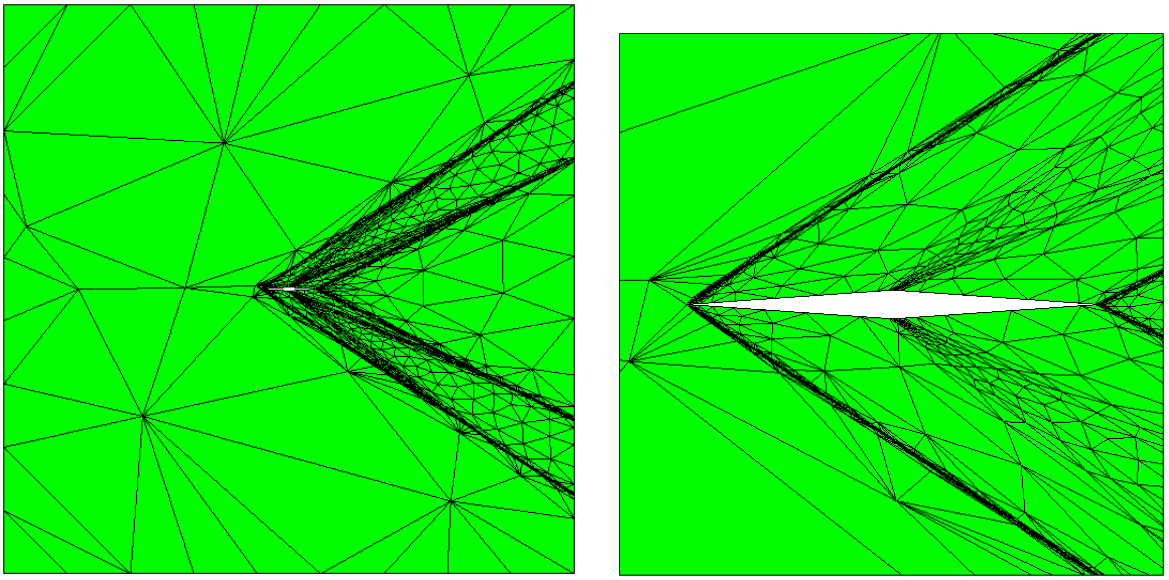
FIGURE 5.9: Mesh configuration 3 initial mesh



(A) Mesh zoomed out, full domain. (B) Mesh zoomed up near the boundary.

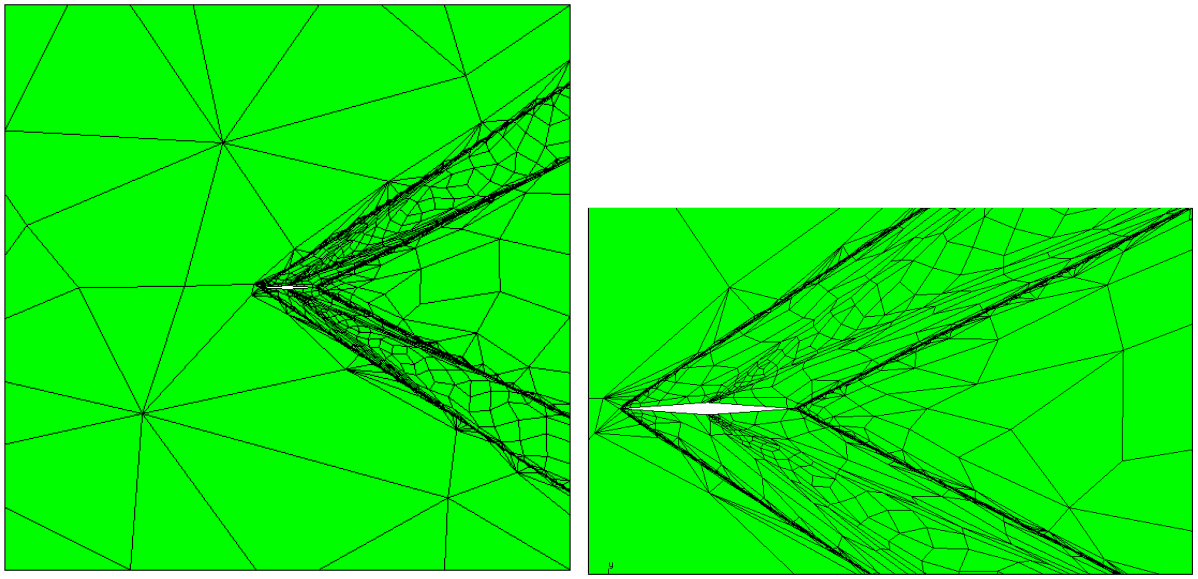
FIGURE 5.10: Mesh configuration 3 recombined mesh.

5.6 Diamond airfoil mesh



(A) Mesh zoomed out, full domain. (B) Mesh zoomed up near the boundary.

FIGURE 5.11: Diamond airfoil initial mesh.



(A) Mesh zoomed out, full domain.

(B) Mesh zoomed up near the boundary.

FIGURE 5.12: Diamond airfoil recombined mesh.

5.7 Cross mesh

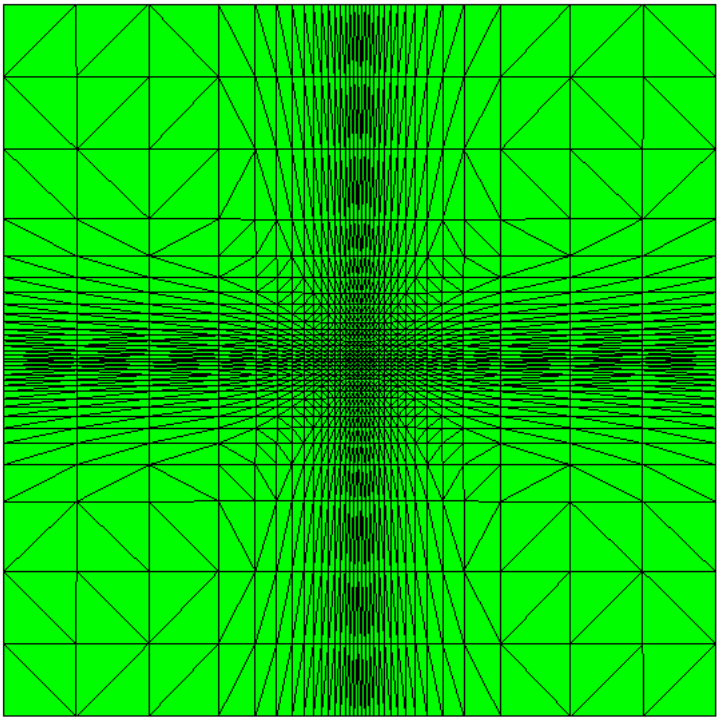


FIGURE 5.13: Quarter circle initial mesh

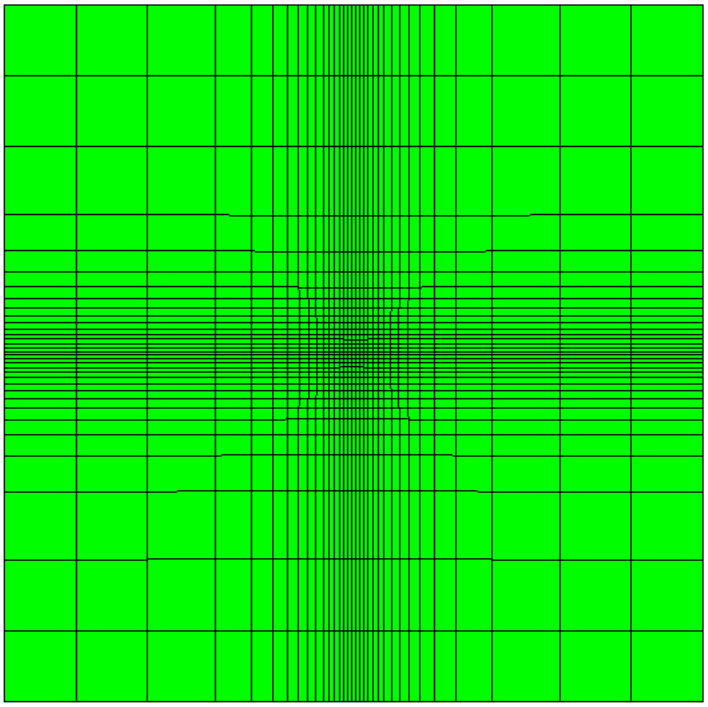


FIGURE 5.14: Quarter circle recombined mesh

5.7.1 Simulation results and data

A simple test case involving scalar Advection-Diffusion in 2D was solved on the mesh as shown in 5.15.

Scalar Advection-Diffusion equation in 2D

$$\frac{\partial w}{\partial x} + \frac{\partial w}{\partial y} - \varepsilon \left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} \right) = s\left(x, y\right), \quad (x, y)\ \epsilon\ \Omega = [0, 1]^2 \tag{5.1}$$

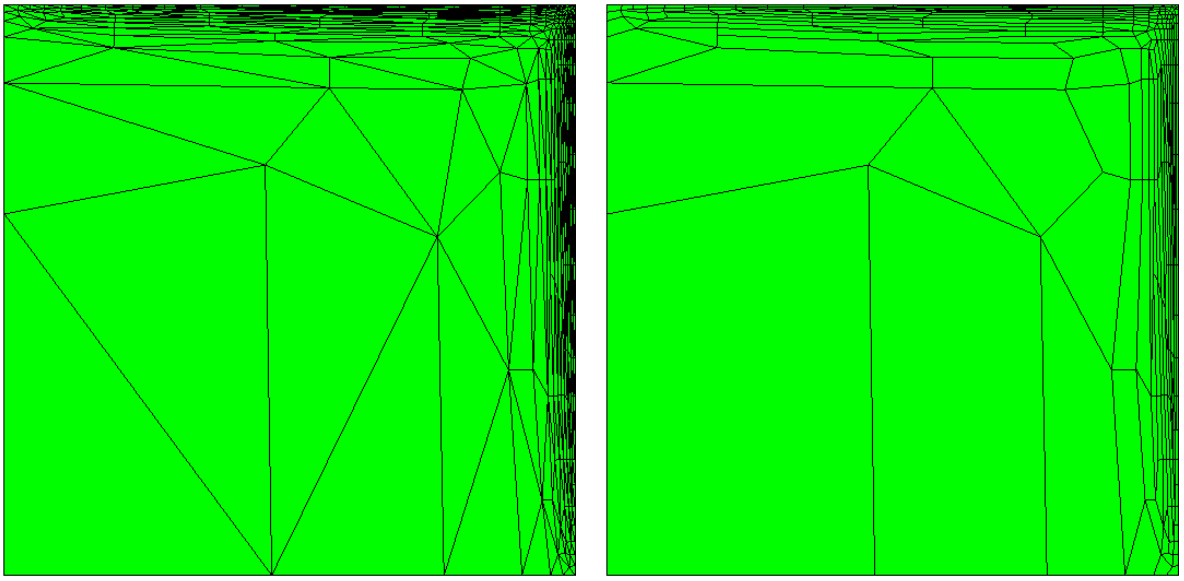
$$w\left(x, y\right) = 0, \quad (x, y)\ \epsilon\ \partial\Omega \tag{5.2}$$

The equation 5.1 represents the Scalar advection-diffusion equation where the first-order partial derivatives represent the *advection* term, and the second-order partial derivatives represent the *diffusion* term. The source term is defined as the $s(x, y)$, ε is the diffusion coefficient. The domain is also set to a 2D square of size 1.

Source term s(x, y) is set such that the solution w(x, y) is given by:

$$w\left(x, y\right) = \left(x + \frac{e^{x/\varepsilon} - 1}{1 - e^{1/\varepsilon}} \right) \cdot \left(y + \frac{e^{y/\varepsilon} - 1}{1 - e^{1/\varepsilon}} \right) \tag{5.3}$$

Where, $\varepsilon = 0.01$

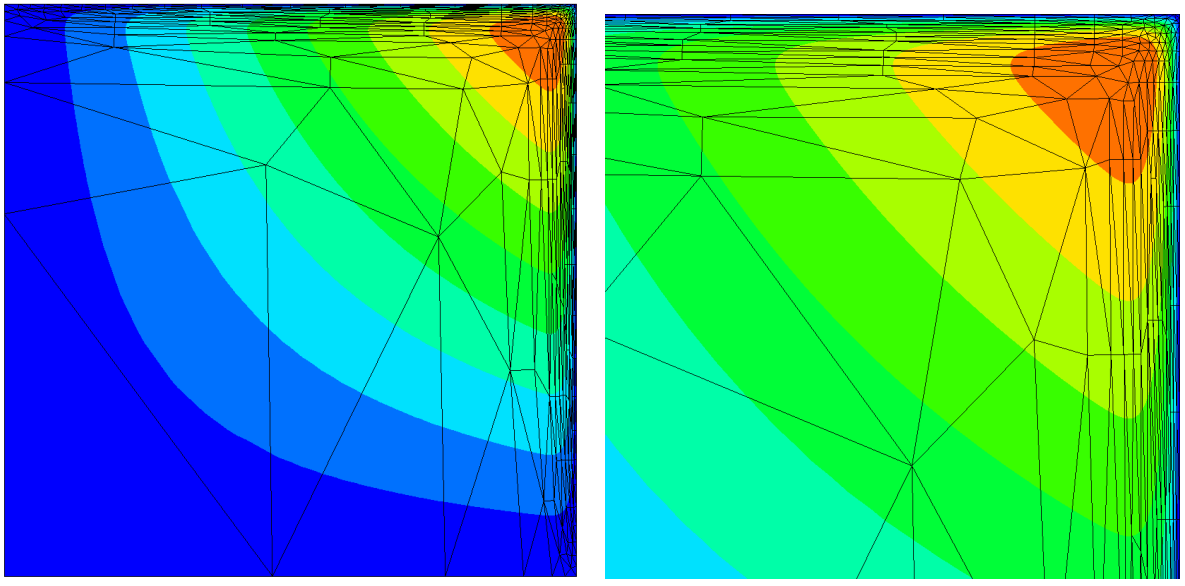


(A) Test mesh - initial.

(B) Test mesh - recombined.

FIGURE 5.15: Test mesh - (A) shows the initial mesh, (B) shows the final recombined mesh.

The simulation was run for both the initial and the recombined mesh meshes. The figure 5.16 shows the contours mapped out for the term w on the initial mesh. The initial mesh consists of *519 triangular elements and 0 quads.*

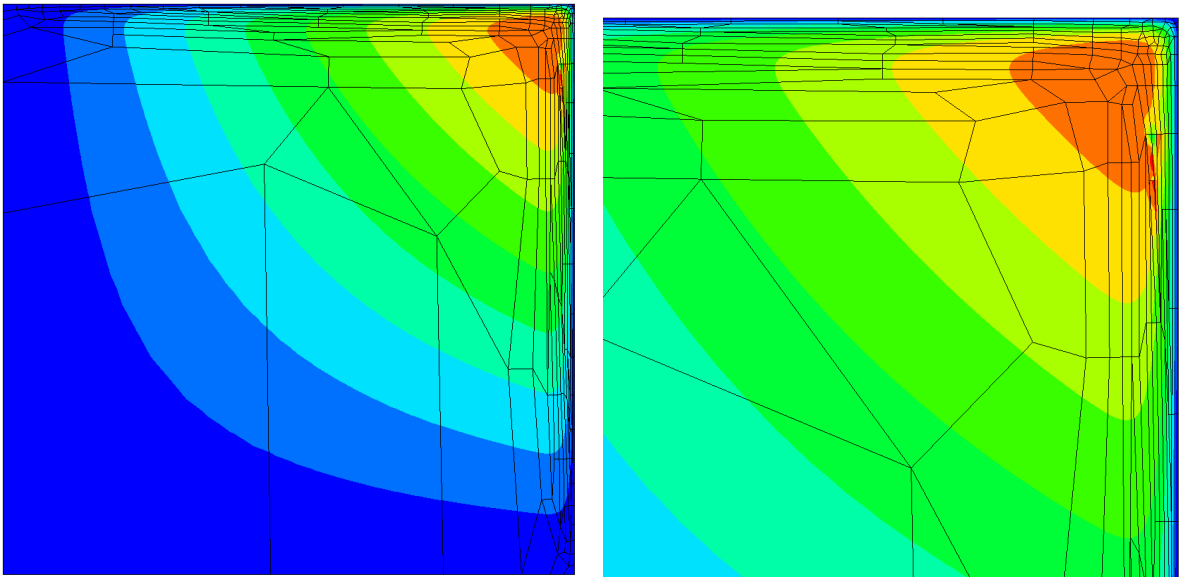


(A) Simulation results for the initial mesh - full domain.

(B) Simulation results for the initial mesh - zoomed in.

FIGURE 5.16: Initial mesh (519 elements) contours for the term "w" - (A) Shows the full domain, (B) Zoomed in near the edge

The figure 5.17 shows the contours mapped out for the term w on the recombined mesh. The recombined mesh consists of *37 triangular elements and 241 quads.* After the



(A) Simulation results for the recombined mesh - full domain.

(B) Simulation results for the recombined mesh - zoomed in.

FIGURE 5.17: Recombined mesh (278 elements) contours for the term "w" - (A) Shows the full domain, (B) Zoomed in near the edge

computation, the following observations were made: From the data as mentioned in the

	Initial mesh	Recombined mesh
Number of triangles	519	37
Number of quads	0	241
Degrees of freedom	3114	2391
Error	2.48119×10^{-5}	4.2466×10^{-4}

TABLE 5.1: Quality metrics for comparison.

table 5.1, the conclusions are:

1. The degrees of freedom decrease in the recombined mesh - the linear systems to solve. This increases computational efficiency and decreases costs.
2. The boundary layer resolution in both cases - the initial mesh and the recombined mesh are the same.
3. The errors were calculated as the $\mathbf{L}^2$ **Norm** of the difference in the numerical result and the analytical result. The errors can be justified by the decrease in the degrees of freedom. As the recombined mesh is refined further, the errors will decrease. This gives us the advantage of faster computations without much loss in accuracy.

Furthermore, this recombination process will be useful, especially in problems simulating turbulent flows where the requirement is to have a structured mesh and elements.

Chapter 6

Parallelization using Message Passing Interface (MPI)

A key strategy in high-performance computing is parallelization, which makes it possible to divide complicated calculations among several processors to increase productivity and shorten execution times. The Message Passing Interface (MPI) is a popular standard for parallelization. MPI enables tasks to be carried out concurrently on several processors by facilitating coordination and communication between processes in a distributed memory environment. It offers an adaptable architecture for message exchange, process synchronization, and data dependency management, which makes it appropriate for a variety of uses, including large-scale data processing and scientific simulations. Developers can maximize scalability and computing performance in distributed systems by utilizing MPI.

6.1 MPI environment setup

A crucial framework for supporting distributed computing is the Message Passing Interface (MPI). The mpi4py package in Python offers a high-level interface for MPI, enabling programmers to effectively implement parallel processing. The following procedures are involved in setting up the MPI environment in a Python program:

1. mpi4py must be installed in the python environment. It provides binding for MPI functions, which then are later used in python scripts.
2. Use the mpi4py library as "from mpi4py import MPI".
3. Declare and set the MPI environment setup shown in the listing 6.1.

(a) **"comm = MPI.COMM_WORLD"** : Represents global communicator which includes all the processes in it.

(b) **"rank = comm.Get_rank()"** : Retrieves the unique process amongst all global processes. This is required to distribute tasks amongst each different processes.

(c) **size = comm.Get_size()"** : Determines the total number of processes that are called by the user, aids in workload distribution.

This configuration serves as the basis for using MPI to implement parallel algorithms. While size makes it easier to divide the workload efficiently, rank aids in assigning certain duties to each process. Processes can communicate with one another using the global communicator (comm), allowing for coordinated calculations throughout the dispersed system.

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```

Listing 6.1: Main body of the program modified for MPI

6.2 Main body - modified for MPI implementation

```
# call the MPI communicators at the start
# REST CODE SAME AS THE SERIAL VERISON

if rank == 0:
    print(f"Element sorting based on quality function has started")


aspect_ratio_max = np.zeros((size,1))

num_per_rank = n_el // size # the floor division // rounds the result down to
    the nearest whole number.
remainder = n_el % size
weight = 0.8

counter = [0 for count in range(size)]

start_time = time.time()
lower_bound = rank*num_per_rank + min(rank,remainder)
upper_bound = lower_bound + num_per_rank + (1 if rank < remainder else 0)

```

```
for el in range(n_el)[lower_bound:upper_bound]:

    neighbor = neighbors_elm(el)

    aspect_ratio_local = np.zeros((3,1))

    for i in range(3):

        if neighbor[i] is not None:
            aspect_ratio = quality_func_aspect_ratio(el, i) # temporarily storing the aspect ratio quality function
            # quality_func = 2
            aspect_ratio_local[i] = aspect_ratio

        else:
            aspect_ratio = 0
            aspect_ratio_local[i] = aspect_ratio

    aspect_ratio_next_el = max(aspect_ratio_local)
    if aspect_ratio_next_el > aspect_ratio_max[rank]:
        aspect_ratio_max[rank] = aspect_ratio_next_el

    del aspect_ratio_local
    gc.collect()

    print(f"Proc no. {rank} ---- Searching for max aspect ratio ---- {(counter[rank]/num_per_rank)*100} % completed", end ='\n')
    counter[rank] += 1

comm.barrier()

# # aspect_ratio_lst = comm.allgather(aspect_ratio_lst_temp)
# # aspect_ratio_lst = [item for sublist in aspect_ratio_lst for item in sublist]

max_at_each_rank = max(aspect_ratio_max)
max_aspect_ratio = comm.allreduce(max_at_each_rank, op=MPI.MAX)

end_time_aspect_ratio = time.time()

if rank == 0:
    print(f"Time taken for aspect ratio list population: {end_time_aspect_ratio - start_time} seconds", flush = True)
    print(f"Aspect ratio quality function has been calculated", flush = True)
    # with open('aspect_ratio_lst.txt', 'w') as filehandle:
    #     for i in range(len(aspect_ratio_lst)):
    #         print(aspect_ratio_lst[i], file=filehandle)
    print(max_aspect_ratio[0])
```

```
64
65 comm.Barrier()
66
67 elm_info_temp = []
68 elm_info = []
69 counter = [0 for count in range(size)]
70
71 for el in range(n_el)[lower_bound:upper_bound]:
72
73     neighbor = neighbors_elm(el)
74
75     for i in range(3):
76
77         if neighbor[i] is not None:
78             edge_vertex = intersection(elements[neighbor[i]][0].vertices,
    elements[el][0].vertices)
79             common_edge = intersection(ma[edge_vertex[0]].edges, ma[edge_vertex
    [1]].edges)
80             quality_func = weight*(quality_func_aspect_ratio(el, i)/
    max_aspect_ratio) + (1-weight)*quality_func_angles(internal_angles(el, i)) #
     ADJUSTING WEIGHTS HERE
81             elm_info_temp.append([el, neighbor[i], i, edges_list[common_edge[0].
    nr][0].nr, quality_func, 1]) # Changing this also would require changes
    below in the recombination part
82
83         elif neighbor[i] is None:
84             edge_vertex = None # This "None" implies that this is a boundary
    edge
85             common_edge = None # This "None" implies that this is a boundary
    edge therefore common_edge is not required
86             # elm_info.append([el, neighbor[i], i, None, quality_func_angles(
    internal_angles(el, i)), 0])
87
88     print(f"Proc no. {rank} ---- Element {el} has been processed ---- {(counter[
    rank]/num_per_rank)*100} % completed", end= '\n')
89     counter[rank] += 1
90
91 comm.barrier()
92
93 elm_info = comm.allgather(elm_info_temp)
94 elm_info = [item for sublist in elm_info for item in sublist]
95
96 end_time_elm_info = time.time()
97
98 if rank == 0:
99     print(f"Time taken for elm_info computation: {end_time_elm_info -
    end_time_aspect_ratio} seconds", flush = True)
100     # with open('quality_check.txt', 'w') as filehandle:
```

```
101     #     for i in range(len(elm_info)):
102     #             print(elm_info[i], file=filehandle)
103
104 for el in ma.Elements(VOL):
105     boundary_weights(el.nr)
106
107 if rank == 0:
108     print(f"Boundary weights have been applied", flush = True)
109
110 elm_info.sort(key = lambda x: x[4], reverse = True)
111
112 # elm_info_for_edge = copy.deepcopy(elm_info)
113 # elm_info_for_edge.sort(key = lambda x: x[0], reverse = False)
114
115 if rank == 0:
116     print(f"Element sorting based on quality function has finished", flush =
    True)
117
118 if rank == 0:
119     with open('quality_check_mpi.txt', 'w') as filehandle:
120         for i in range(len(elm_info)):
121             print(elm_info[i], file=filehandle)
122
123 start_time_recombine = time.time()
124
125 num_per_rank = len(elm_info) // size # the floor division // rounds the result
    down to the nearest whole number.
126 lower_bound = rank*num_per_rank + min(rank,remainder)
127 upper_bound = lower_bound + num_per_rank + (1 if rank < remainder else 0)
128 data_for_recombine = []
129
130 elm_info = np.array(elm_info, dtype=object) # Converting python list to numpy
    array
131
132 if rank == 0:
133     iterations = 0
134     for recombine in range(len(elm_info)):
135
136         curr_el_nr = elm_info[recombine][0]
137         curr_el_neighbor_nr = elm_info[recombine][1]
138         elm_edge_nr = elm_info[recombine][2]
139         edge_info = elm_info[recombine][3]
140         edge_activity = elm_info[curr_el_nr][5]
141
142         v0_nr = neighbors_vert(curr_el_nr)[1][0]
143         v1_nr = neighbors_vert(curr_el_nr)[1][1]
144         v2_nr = neighbors_vert(curr_el_nr)[1][2]
145
```

```
146         if element_activity[curr_el_nr] == 1:
147 
148             if elm_edge_nr == 0:
149                 if element_activity[curr_el_neighbor_nr] == 1:
150                     # print("2"," ","1", " ", "0", " ", "0", "4", v0_nr+1, v1_nr
    +1, neighbors_vert(curr_el_nr)[2][0].nr+1, v2_nr+1, file=filehandle)
151                     data_for_recombine.append([2, 1, 0, 0, 4, v0_nr+1, v1_nr+1,
    neighbors_vert(curr_el_nr)[2][0].nr+1, v2_nr+1])
152 
153                     element_activity[curr_el_nr] = 0
154                     element_activity[curr_el_neighbor_nr] = 0
155 
156             if elm_edge_nr == 1:
157                 if element_activity[curr_el_neighbor_nr] == 1:
158                     # print("2"," ","1", " ", "0", " ", "0", "4", v0_nr+1, v1_nr
    +1, v2_nr+1, neighbors_vert(curr_el_nr)[2][1].nr+1, file=filehandle)
159                     data_for_recombine.append([2, 1, 0, 0, 4, v0_nr+1, v1_nr+1,
    v2_nr+1, neighbors_vert(curr_el_nr)[2][1].nr+1])
160 
161                     element_activity[curr_el_nr] = 0
162                     element_activity[curr_el_neighbor_nr] = 0
163 
164 
165             if elm_edge_nr == 2:
166                 if element_activity[curr_el_neighbor_nr] == 1:
167                     # print("2"," ","1", " ", "0", " ", "0", "4", v0_nr+1,
    neighbors_vert(curr_el_nr)[2][2].nr+1, v1_nr+1, v2_nr+1, file=filehandle)
168                     data_for_recombine.append([2, 1, 0, 0, 4, v0_nr+1,
    neighbors_vert(curr_el_nr)[2][2].nr+1, v1_nr+1, v2_nr+1])
169 
170                     element_activity[curr_el_nr] = 0
171                     element_activity[curr_el_neighbor_nr] = 0
172 
173         percentage = ((iterations+1)/len(elm_info))*100
174         print(f"{iterations} : Element {curr_el_nr} and Element {
    curr_el_neighbor_nr} recombined ---- on Proc. {rank} ---- {percentage} %
    completed", end = '\n')
175         iterations += 1
176 
177     elms_recombined = iterations
178 
179     isolated_triangles = 0
180     iterations_island_triangles = iterations
181     for el in ma.Elements(VOL):
182 
183         if element_activity[el.nr] == 1:
184 
185             isolated_triangles += 1
```

```
186
187             v0_nr = neighbors_vert(el.nr)[1][0]
188             v1_nr = neighbors_vert(el.nr)[1][1]
189             v2_nr = neighbors_vert(el.nr)[1][2]
190
191             # print("2"," ","1", " ", "0", " ", "0", "3", v0_nr+1, v1_nr+1,
    v2_nr+1, file=filehandle)
192             data_for_recombine.append([2, 1, 0, 0, 3, v0_nr+1, v1_nr+1, v2_nr
    +1])
193
194             # percentage = ((elms_recombined*2 + isolated_triangles)/n_el)*100
195             print(f"{iterations_island_triangles} : Element {el.nr} is an island
     triangle")
196             iterations_island_triangles += 1
197
198 with open(f"recombined_elements{weight}.txt", 'w') as filehandle:
199     for i in range(len(data_for_recombine)):
200         print(*data_for_recombine[i], file=filehandle)
201
202 end_time_recombine = time.time()
203
204 print(f"Time taken for aspect ratio list population: {end_time_aspect_ratio -
    start_time} seconds   |   Time taken for elm_info computation: {
    end_time_elm_info - end_time_aspect_ratio} seconds   |   Time taken for
    recombination: {end_time_recombine - start_time_recombine} seconds", flush =
     True)
```

Listing 6.2: Main body of the program modified for MPI

In the listing 6.2, the workload distribution is assigned to each process based on its rank. *"lower_bound"* and *"upper_bound"* are the variables that define the portion of elements each process will handle. They are computed using the total number of processes (size), the rank of each process (rank), and the total number of elements (n_el). This will ensure an *almost* equal workload distribution. There are also instances where there are *"if statements"* conjoined with *"rank == 0"*, this effectively asks the program to do that block of code from the program on the master process (Serially). Implementing MPI significantly speeds-up the program, achieving speedups of upto 15x.

6.3 Bash script for automation - modified for MPI implementation

The bash script mentioned in the listing 6.4 is similar to the bash script mentioned in listing 4.8 mentioned in **section 4.8 of chapter 4**, with the only difference being how

the program is compiled and run inside the bash script or on bash (linux terminal) for that matter.

```
mpirun -n ‘‘size’’ python ‘‘program.py’’
```

Listing 6.3: Command to run Parallely using MPI

In the listing 6.3, replace “size” with the number of processes that are needed and “program.py” with the file name of your script *(without the quotes “”)*.

```
#!/bin/bash

rm -rf vol_files
mkdir vol_files
touch mesh_quad_original.vol
cp scalarBoundaryLayer.vol mesh_quad_original.vol # change the first name of the file accordingly to your mesh file

for k in 0.8 # looping over different weights
do

    rm -f recombined_elements${k}.txt

    # the numbers 411, 460,538 are the line numbers in the python program where weight and file names are extracted from. If code is changed, change these numbers accordingly and the file names as your python program name
    sed -i "410s/weight = .*/weight = ${k}/g" quad_max.py
    sed -i "466s/with open('recombined_elements.*.txt', 'a') as filehandle:/with open('recombined_elements${k}.txt', 'a') as filehandle:/g" quad_max.py

    echo "Running for weight = ${k}"

    mpirun -n 5 python quad_max.py # running the python program

    if [ $? -eq 0 ]; then
    echo "Success"
    else
    echo "Failed"
    fi

    # a=$(grep "surfaceelements" "mesh_quad_original.vol" | cut -d: -f1)
    start_line=$(cat -n mesh_quad_original.vol | grep "surfaceelements" | awk '{print $1}') #start line
    end_line=$(cat -n mesh_quad_original.vol | grep "volumeelements" | awk '{print $1}') #end line

    # geabbing the start and end lines in file to copy
```

```
32      c=$(expr $start_line)
33      d=$(expr $end_line - 2)
34
35      total_elements=$(wc -l recombined_elements${k}.txt | awk '{print $1}')
36
37      touch mesh_quad_${k}.vol
38      cp mesh_quad_original.vol mesh_quad_${k}.vol
39
40      head -n $((c)) "mesh_quad_${k}.vol" > temp_file.txt
41
42      echo $((total_elements)) >> temp_file.txt
43
44      cat "recombined_elements${k}.txt" >> temp_file.txt
45      tail -n +$((d)) "mesh_quad_${k}.vol" >> temp_file.txt
46      mv temp_file.txt "mesh_quad_${k}.vol"
47
48      mv mesh_quad_${k}.vol vol_files/ # moving the files to a new directory
49
50      rm -f recombined_elements${k}.txt # deleting the temporary files
51
52  done
53
54  rm -f mesh_quad_original.vol
```

Listing 6.4: Bash script for automation - Modified for MPI implementation

Chapter 7

Conclusion and Future work

Throughout this thesis, mesh recombination and its processes are discussed. We start with a triangular mesh, which is used as an input for the recombination program which then runs and outputs the final quad-dominant mesh. This final quad-dominant mesh, obtained after the recombination program, would prove useful in obtaining higher accuracy in simulations as quad elements align better with flow features such as direction or geometrical features, more specifically, in problems involving boundary layers or compressible flows.

7.0.1 Sorting methods

The aspect ratio of all possible quads that could form with the elements was calculated. Another quality criterion, referred to as "edge quality," was also calculated. A linear combination of these two criteria was taken and then sorted in a decreasing fashion. Post this sorting, the boundary elements - either to the domain or the geometry were multiplied by a weight; this was done so to provide them with the highest priority in the recombination program.

7.0.2 Population of elm_info array

An array was populated with all the information required by the recombination program, namely - the current element, neighbours to the current element, edge number, edges list, and quality function values. This array was looped around for writing in the file for the output recombined mesh. It was also observed that the array population or the aspect ratio calculations were slow, and it was essential to speed up the recombination program. Hence, parallelization was implemented. Message passing interface (MPI), which is a distributed memory method for parallelizing the program, was chosen.

7.0.3 Future work

In conclusion, the program is efficient in the recombination process. To make it more specific to "quad meshes or hybrid meshes", vertex smoothing techniques can be used - some of them which include:

1. Laplacian smoothing
2. Angle based smoothing

Plans for future work regarding the program have been laid out as mentioned below:

1. Vertex smoothing techniques shall be used, with more preference and *bias-ness* towards "Angle based smoothing" since, "Laplacian smoothing" is not very effective for meshes having very skewed elements.
2. The program will be optimized further using some C/C++ based python libraries such as Numba which is a high performance python compiler or pybind11 which is a lightweight header-only library that exposes C++ types in Python and vice versa.
3. Simulating the problems involving turbulence using our recombined mesh.
4. Extending the program to 3D meshed geometries, expanding the usability in real life simulations and scenarios.

Bibliography

[1] Frederic Alauzet. "Metric-Based Anisotropic Mesh Adaptation". In: (2010).

[2] Houman Borouchaki and Pascal Frey. *Adaptive Triangular-Quadrilateral Mesh Generation*. Research Report RR-2960. INRIA, 1996. URL: https://hal.inria.fr/inria-00073738.

[3] Ghalia Guiza et al. *Anisotropic boundary layer mesh generation for reliable 3D unsteady RANS simulations*. Mar. 2019.

[4] Adrien Loseille and Frédéric Alauzet. "Continuous Mesh Framework Part I: Well-Posed Continuous Interpolation Error". In: *SIAM J. Numerical Analysis* 49 (Jan. 2011), pp. 38–60. DOI: 10.1137/090754078.

[5] Keigan MacLean and Siva Nadarajah. "Anisotropic mesh generation and adaptation for quads using the Lp-CVT method". In: *Journal of Computational Physics* 470 (2022), p. 111578. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2022.111578. URL: https://www.sciencedirect.com/science/article/pii/S0021999122006404.

[6] A. E. PUCKETT and H. J. STEWART. "THE THICKNESS OF A SHOCK WAVE IN AIR". In: *Quarterly of Applied Mathematics* 7.4 (1950), pp. 457–463. ISSN: 0033569X, 15524485. URL: http://www.jstor.org/stable/43633762 (visited on 10/11/2024).

[7] J.-F. Remacle et al. "Blossom-Quad: A non-uniform quadrilateral mesh generator using a minimum-cost perfect-matching algorithm". In: *International Journal for Numerical Methods in Engineering* 89.9 (2012), pp. 1102–1119. DOI: https://doi.org/10.1002/nme.3279. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.3279. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.3279.