Budapest University of Technology and Economics
Faculty of Mechanical Engineering

Department of Fluid Mechanics

Automated CFD mesh generation using machine learning

MASTER'S THESIS

Author:
Hajdú Bence

Supervisor:
Dr. Josh Davidson
Research Fellow

Budapest, 2021

Declarations

Declaration of acceptance

This thesis fulfills all formal and content requirements prescribed by the Faculty of Mechanical Engineering of Budapest University of Technology and Economics, as well as it fully complies all tasks specified in the transcript. I consider this thesis as it is suitable for submission for public review and for public presentation.

Done at Budapest, 13.12.2019

Dr. Josh Davidson

Declaration of independent work

I, Bence Hajdú, the undersigned, hereby declare that the present thesis work has been prepared by myself without any unauthorized help or assistance such that only the specified sources (references, tools, etc.) were used. All parts taken from other sources word by word or after rephrasing but with identical meaning were unambiguously identified with explicit reference to the sources utilized.

Done at Budapest, 28.05.2021

Hajdú Bence

Abstract

In recent decades, CFD has been used extensively both in the industry and in the research field, but it's still facing many difficulties. Perhaps one of the most notable difficulty is the automation of the numerical mesh generation. Current meshing algorithms only considers the topology of the geometry, therefore engineers have to take into account the flow characteristics themselves and modify the mesh accordingly.

There have been several publications in the nineties and the 2000s where machine learning has been applied to numerical mesh generation or mesh optimization. These approaches have few downsides. They often require some sort of preprocessing, and most of them use meshes as training data. Using a diverse training data set with good quality is crucial for training a machine learning model, since its performance is highly dependent on is. With the usage of a physics informed neural network the need for meshes to generate training data could be prevented. The model can predict the solution of a wide variety of differential equations, thus can be used to approximate a flow field. The approximate then can be combined with a meshing algorithm to produce a graded mesh which considers physical properties alongside the domain's properties

In this work the variational tetrahedral meshing algorithm by Aliez et al. was combined with the physics informed neural network to produce mesh grading. The algorithm can adjust the mesh density utilizing a weight function. The idea is to compute a weight value for each node in the function of the predicted flow field, the larger the weight the denser the mesh be around that node. The paper describes a geometry-optimal function, but it is also possible to implement other functions.

Keywords: *Physics informed neural networks, Delaunay triangulation*

Contents

Chapter 1

Introduction

1.1 Computational fluid dynamics

Computational Fluid Dynamics (CFD) had long been used by engineers to reduce the reliance on physical model tests in the design process, thus reducing the development cost. The effectiveness of a CFD simulation relies on an appropriate trade-off between the complexity of the simulation and the accuracy of the solution [1].

In the beginning CFD codes were written for specific problems, and were mainly used by the energy sector for modeling furnaces and combustion chambers [Khalil 2012]. Later on it was recognized that these codes could be generalized, this enabled the use of CFD in a wide variety of disciplines and industries [2]. Today CFD industrial usage is including but not limited to:

- Aerodynamic simulation of aerospace flights and shuttles as well as other aerospace applications
- automotive industry, for simulating vehicles component's thermal and aerodynamical properties
- Pharmaceutical industry, for designing chemical reactors [3]
- In medical research for designing of drug delivery systems [4]
- It is also used in the design of electronics, and batteries to optimize thermal properties [5].

To apply a general-purpose CFD code to a problem, the domain must be defined. It is done via computer assisted design sofwares (CAD). Then the domain needs to be discretised because the governing equations are solved trough means of numerical methods. The discretization of the domain is called meshing or grid generation. Over 50 % of the time spent on an industrial CFD project is devoted to the domain generation and meshing, and since the accuracy and the computational cost of a simulation is highly dependent on the mesh, these steps are generally the most important process in the pipeline [6].

Since a fully automated mesh generator is not available, therefore an experienced analyst is required to produce a credible mesh for a complex problem making this process even more costly. Kim et. al. summarised the cost of a CFD simulation of Numerical water basin (NWB) problems [1]. They also found that half of the cost of a CFD project's is the man hours. Figure 1.1 shows the composition of a NWB simulation's cost.

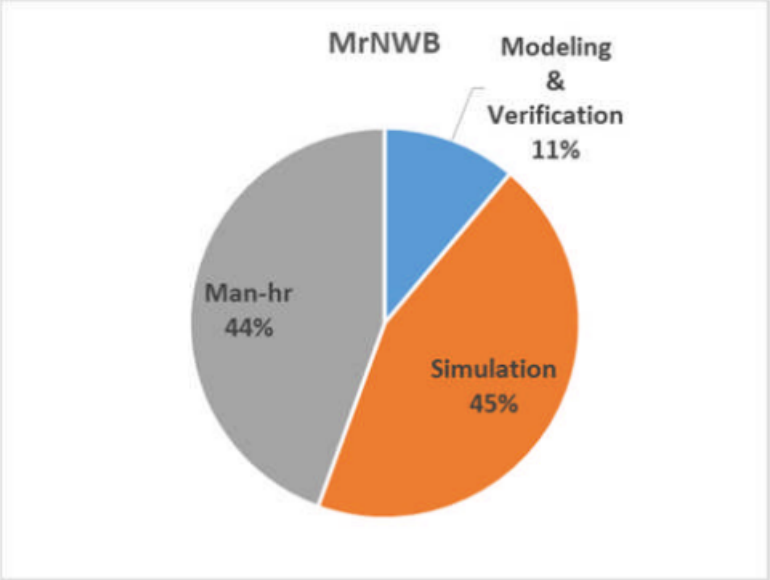

Figure 1.1: Cost of NWB CFD simulation [1]

Figure 1.2 is to demonstrate why does meshing requires high level of expertise. A domain what is used for meshing consists of curved and flat surfaces, sharp edges and small gaps. To get an accurate simulation result the meshing engineer needs to have a decent guess about the flow field in advance, to be able to decide where are the points of interests, thus where the mesh needs refinements. Refinements are important because where the mesh is denser the simulation results will be more accurate, but in return it increases the computational cost [7].

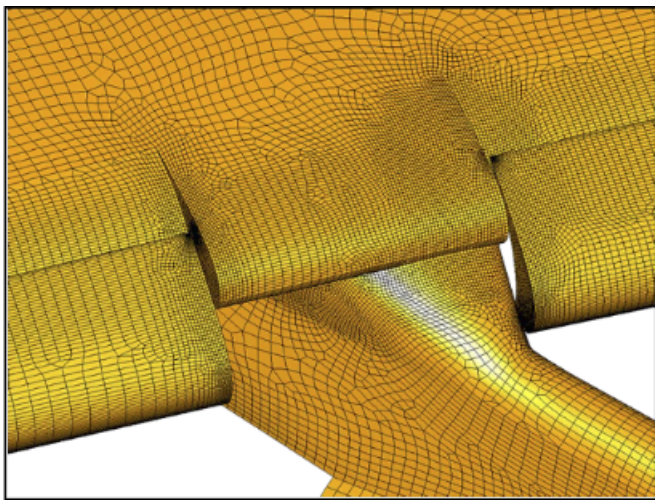

Figure 1.2: Pylon intersection part of a wing [7]

Since there are no fixed rules for deciding which are the points of interest, finding

a method that combines features, such as low computational cost (time and resources), an acceptable mesh quality, the number of points to be used and the ability to model structures with complex geometry is not an easy task. This is specifically true because a certain mesh and cell quality is required in order to diminish the numerical error being produced by the discretization process [8]. Artificial Neural networks (ANN) have shown to be particularly powerful in dealing with high-dimensional data and modeling nonlinear and complex relationships thus it's application to similarly complex problems are widely used nowadays [9].

1.2 Machine learning

Machine learning has gotten popular in the last decade, but surprisingly the first learning model, the perceptron was created in 1958 by Rosenblatt [10]. The next major advance came in 1986 when Rumbelhart made a multi layer model (they used multiple perceptrons together), but they struggled with hardware limitations [11]. ANNs as we know it today was enabled by the rapidly evolving computational power. The increasing computational power not only made possible to calculate faster, but to record and store large amount of data. While traditional programs processes the data along a predefined rule set, on the contrary machine learning automatically formulates the rules from the input data figure 1.3.

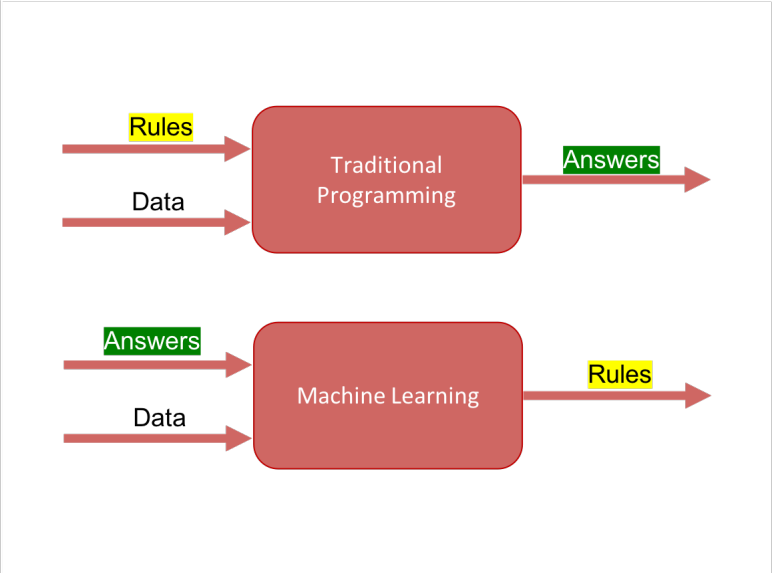

Figure 1.3: Illustration of the fundamemntal difference between a traditional algorithm and a machine learning algorithm. [12]

Today, ANNs are applied in many fields, in retail it is used to optimize inventory management and for targeted advertisement, used for weather forecasting and for fraud detection to name a few [13]. Although not many machine learning application used in the engineering field but recent development of self-driving cars has showed that it is suitable solution for mission critical applications as well.

1.3 Thesis objectives

This thesis will investigate the possibilities of the automatization of CFD meshing with the aid of machine learning. A meshing algorithm will be implemented in python, then the program will be complemented with a machine learning model. To validate the produced mesh, OpenFOAM's lid driven cavity simulation will be used.

The machine learning model's main task will be to provide a density filed for the meshing program in the function of different physical properties, thus the algorithm can make a sizing field what respects the flow field besides the geometry. To find the most suitable machine learning model an extensive literature review in the subject of automated meshing with machine learning will be carried out as well.

1.4 Thesis outline

Chapter 2 presents a theoretical review of computational fluid dynamics, Delaunay triangulation, Machine Learning, physics informed neural networks and reviews the publications found in the topic of machine learning aided numerical mesh generation.
Chapter 3 discusses the tools used in this work; OpenFOAM a library for CFD, used python packages and scripts, DeepXDE a frame work for implementing physics informed neural networks.
Chapter 4 presents the results achieved in this work.
Chapter 5 presents the conclusions found during the present study, along with potential future works.

Chapter 2

Literature review

The literature relating to automatic CFD generation is reviewed in this chapter. Section 2.1 presents the governing equations of CFD simulations, and their discretization. Section 2.2 introduces some basic mesh quality measures, and talks about triangular meshes, and how can they be optimized. Section 2.3 lays down the basic of machine learning. Section 2.4 is a miniature review of the published works in machine learning aided numerical mesh generation. Section 2.5 presents the Physics Informed Neural Networks.

2.1 Computational Fluid Dynamics

To significantly reduce the computational cost in continuous-based physics like fluid dynamics, instead of considering the molecular structures of the system, we use average properties function of time and space. Such numerical method is CFD, where we segregate space using a grid [14][Frank2020].

CFD simulations can be divided into three major stages[15]:

1. Pre-processing, where the definition and the discretization of the computational domain (the region of interest) happens. During discretization the domain gets divided into a number of smaller, non overlapping sub-domains, which are called control volumes or cells. The "output" of this stage is a mesh.

2. Simulation phase, where the simulation gets set up by selecting the appropriate physical and numerical model. Most of the commercial CFD softwares use the numerical solution method called finite volume method.

3. Post-processing, where the results get visualized, and the simulation model gets validated.

2.1.1 Navier-Stokes Equation

The objective of a fluid simulation is to calculate the velocity and the pressure values in a control volume [16]. To do so the equations of fluid motion have to be defined. The

governing equations of a fluid flow are called the Navier-Stokes equations (NS). NS consist of three equations [17]:

1. Law of conservation of mass equation 2.1, the mass of entering a fluid element must be equal to those that are leaving the same element.

2. Newton's second law of motion equation 2.2, the external forces acting on a fluid particle is equal to its rate of change of linear momentum.

3. First law of thermodynamics equation 2.3, the rate of change of energy of a fluid particle is equal to the heat addition and the work done on the particle [18].

$$\frac{\partial \rho}{\partial t} + \rho \frac{\partial U_i}{\partial x_i} = 0 \tag{2.1}$$

$$\rho \frac{\partial U_j}{\partial t} + \rho U_i \frac{\partial U_j}{\partial x_i} = -\frac{\partial P}{\partial x_j} - \frac{\partial \tau_{ij}}{\partial x_i} + \rho f \tag{2.2}$$

$$\rho c_\mu \frac{\partial T}{\partial t} + \rho c_\mu \frac{\partial T}{\partial x_i} = -P \frac{\partial U_i}{\partial x_i} + \lambda \frac{\partial^2 T}{\partial x_i^2} - \tau_{ij} \frac{\partial U_j}{\partial x_i} \tag{2.3}$$

Where t is time, U is the velocity, x_i and x_j are the Cartesian coordinate components, ρ is the density, τ_{ij} is the viscous stress tensor equation 2.4, f are the external body forces (e.g. gravitational force), δ_{ij} denotes is the Kronecker delta and μ is the dynamic viscosity of the fluid.

$$\tau_{ij} = -\mu \left(\frac{\partial U_j}{\partial x_i} + \frac{\partial U_i}{\partial x_j} \right) + \frac{2}{3} \delta_{ij} \mu \frac{\partial U_k}{\partial x_k} \tag{2.4}$$

These equations can be simplified if incompressible fluid is assumed, the temperature is constant and the external body forces are negligible, applying the divergence theorem we can rewrite eq. 2.1 and eq. 2.2 into eq. 2.5 and eq. 2.6.

$$\nabla \cdot (\rho U) = 0 \tag{2.5}$$

$$\nabla \cdot (\rho U U) = -\nabla P + \mu \cdot \nabla^2 U \tag{2.6}$$

2.1.2 Finite volume method

The NS are analytical equations, and they do not have an exact solution, therefore to solve it numerical methods are required. To do so, the equations have to be discretised to algebraic equations [16]. Such discretization method is the finite volume method (FVM), where the physical space gets divided into small sub-domains, called control volumes or cells, then the partial differential equations get recast on these cells and they are approximated by the nodal or central values of the control volumes [19]. FVM's advantage is that it's conservative. If in each cell of the domain the NS is satisfied, then it will be automatically satisfied for the whole domain [20].

FVM is based on the integral form of the conservation laws. By doing the volume integrals of equations 2.5 and 2.6 and casting Gauss's theorem we get [19]:

$$\sum_i \left(F_i u\right)_{face} = \left(\overline{\frac{\partial p}{\partial x}} V\right)_{cell} + \mu \sum_i \left(\nabla u \hat{n}_i S_i\right)_{face} \tag{2.7}$$

$$\sum_i \left(F_i v\right)_{face} = \left(\overline{\frac{\partial p}{\partial y}} V\right)_{cell} + \mu \sum_i \left(\nabla v \hat{n}_i S_i\right)_{face} \tag{2.8}$$

$$\sum_i \left(F_i w\right)_{face} = \left(\overline{\frac{\partial p}{\partial z}} V\right)_{cell} + \mu \sum_i \left(\nabla w \hat{n}_i S_i\right)_{face} \tag{2.9}$$

$$\sum_i \left(F_i\right)_{face} = 0 \tag{2.10}$$

Where S_i is the surface area of face i, n_i is the normal vector of face S_i, $F_i = \rho U n_i S_i$ is the mass flux going trough face i of the control volume and V is the volume of the control volume. The right hand sides of eqs. 2.7-2.9 are the pressure term and the diffusive terms respectively, and on the left hand side is the convective terms.

The equations above are applied to each cell in the domain. The calculated pressures are typically stored in the centre of the cells, the velocity components are either stored in the centre or at the faces of the cells. To calculate every variable in each cell iterative algorithms are used, such method is the SIMPLE algorithm, what stands for Semi-Implicit Method for Pressure Linked Equations. It's schematics can be seen in figure 2.1. It starts with an initial guess for P and U, then from these calculates a pressure correction quantity P' which then used for recalculating the transport quantities. If the new values satisfy some convergence criterion, then the iteration stops, else these values will be the new inputs of the SIMPLE algorithm.

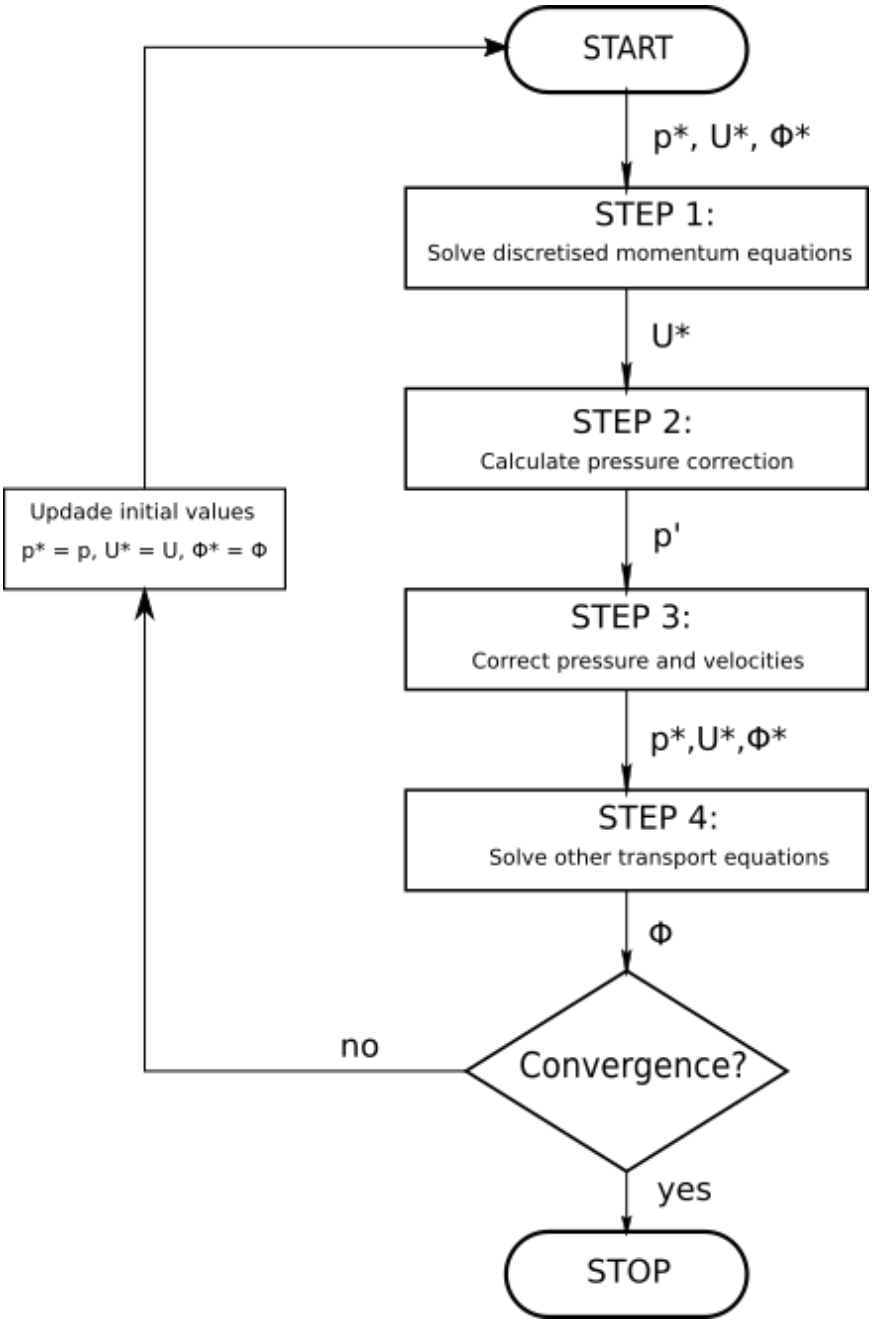

Figure 2.1: SIMPLE algorithm's steps in a block diagram. p U and ϕ denotes the pressure velocity and other transport quantities. [14]

2.2 Meshes

Finite volume meshes are of two basic types: structured and unstructured. In a (2D) structured mesh cells are referenced by an ordered pair of indices. The benefit of using a structured mesh is that the indices of adjoining cells are automatically known and data may be held in 2D arrays efficiently [21].

In an unstructured mesh cells are referenced by a single index. The locations of adjoining cells must be stored in more complicated data structures. But for complex geometries, unstructured meshes are easier to generate [21]. Unstructured 2D meshes usually consist

of triangular cells. In the next few pages I am going to talk about unstructured triangular meshes.

2.2.1 Mesh quality

The ideal mesh of nearly regular simplices (triangles or tetrahedra), and in practice it is expected that we can control the resolution of the mesh locally. Unfortunately there is no general best quality measurement exists for an element, because it depends on the numerical method used and on the problem being solved [22].

The most widely used quality measures are smoothness, skewness and aspect ratio [23] . The same attributes are considered in both 2D and 3D meshes. These quality attributes will be presented trough triangular cells, but the same principles apply for tetrahedral elements as well.

Smoothness is defined as volume ratio of two cells adjacent to the same inner face. The smoothness lies between 0 and 1, and the lager it's value the better the mesh's quality, examples can be seen on figure 2.2 [23].

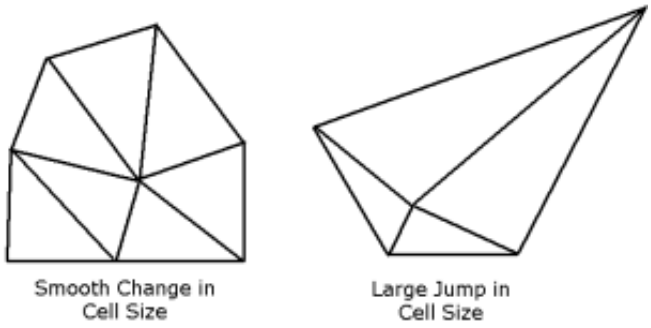

Figure 2.2: A mesh with high smoothness value on the right and a mesh with low smoothness value on the left [23].

Skewness is defined as the difference between the shape of the cell and the shape of an equilateral cell of equivalent volume. If the cell's skewness is 0 then it is equilateral, if it's 0 then it's a completely degenerate cell shown in figure 2.3. Degenerate cells (slivers) are characterized by nodes that are nearly coplanar (collinear in 2D). Highly skewed faces and cells are unacceptable because the equations being solved assume that the cells are relatively equilateral/equiangular. There are many method to calculate an element's skewness, eq. 2.11 is used to calculate the equilateral volume based skewness [23].

$$Skewness = \frac{Optimal\ cell\ size - Cell\ size}{Optimal\ cell\ size} \tag{2.11}$$

Where the optimal cell size is the size of an equilateral cell with the same circumradius.

Highly skewed cells can decrease accuracy and destabilize the solution. Figure 2.4 demonstrates the consequences of using a bad mesh during a simulation [24].

In figure 2.5 few of the most common attributes of a triangle cell are shown. The

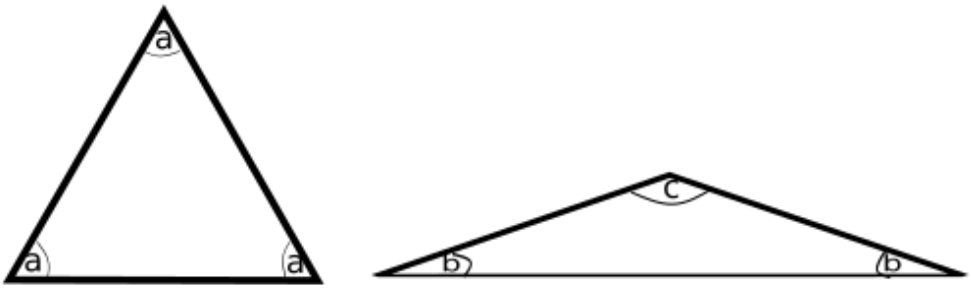

Figure 2.3: On the right there is an equilateral cell, on the right a highly skewed element.

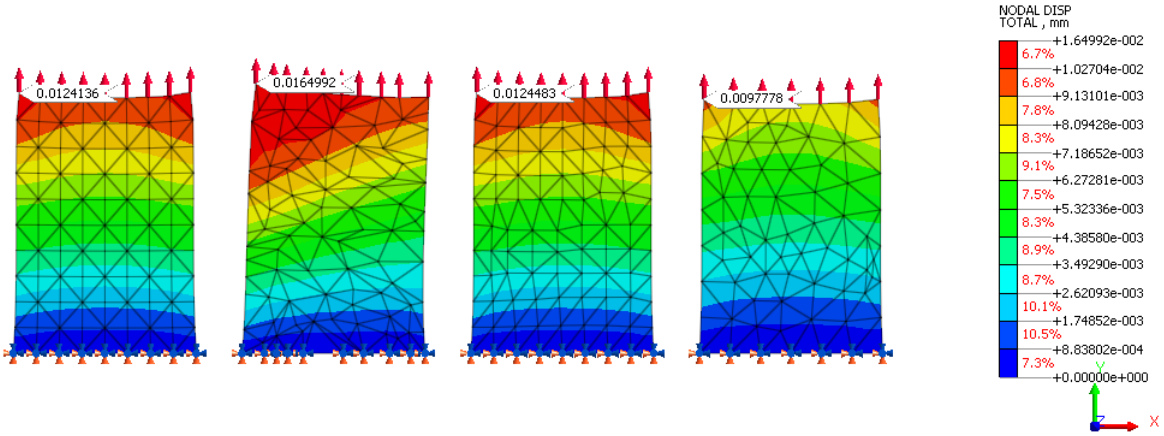

Figure 2.4: Consequences of mesh quality in the same finite element simulation. Four plates of 1mm thickness, same boundary conditions, all fixed at the bottom, all are loaded with the same 200N force applied at the top. [24]

aspect ratio of a cell can be expressed with the specific ratios of these values [25]. Few of the most commonly used aspect ratios are:

- The smallest angle (α)
- The ratio of the radius of circumcircle and the innercircle ($\frac{r}{R}$)
- The ratio of the longest edge and the height ($\frac{max(a,b,c)}{m}$)
- The ratio of the shortest edge and the radius of the circumcircle ($\frac{min(a,b,c)}{R}$)

It can be seen that an equilateral triangle satisfies these attributes the best.

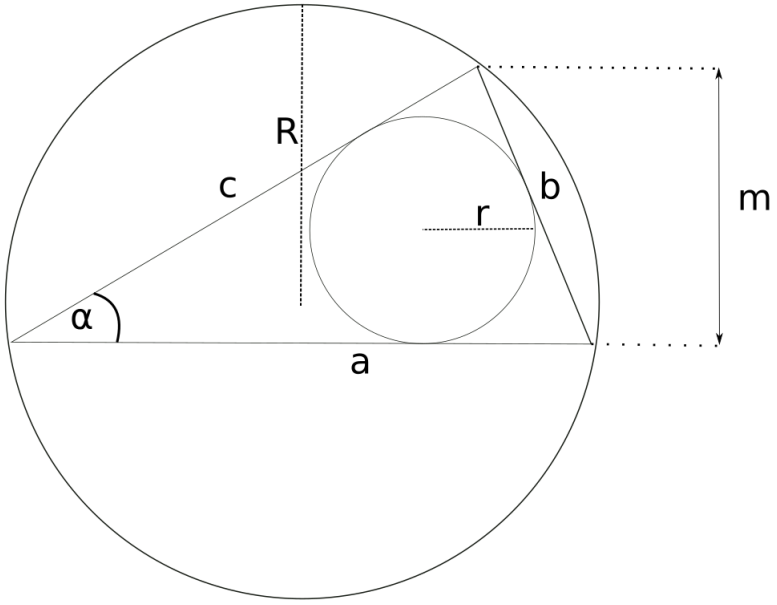

Figure 2.5: Typical attributes of a triangular mesh cell

2.2.2 Delaunay triangluation

Although there are countless triangulation methods, most of them are based on heuristics or has only very elementary guarantees (e.g. that the mesh is meaningful, not degenerate, etc.). The best known method with theoretical guarantees is the Delaunay triangulation, which is for a fixed set of points is optimal according to many different criteria. Main properties of the Delaunay triangulation [25]:

- The centre of the circumscribable circle of each triangle is located inside the triangle.
- Maximizes the smallest angle
- Minimalizes the radius of the circumcircle
- Minimalizes the sum of the incircles radius
- It is exact if there are no quadrilaterals

A very important property of a Delaunay mesh is that it is dual with Voronoi diagram. Voronoi diagram partitions a plane with n points into convex polygons such that each polygon contains exactly one generating point and every point in a given polygon is closer to its generating point than to any other's. For the same set of points there is one Voronoi tessellation and one Delaunay triangulation, that's why they are dual to each other [26]. Figure 2.6 demonstrates the described dual property. Each vornoi cell's corner is the middle point of a delaunay triangle's circumcenter, and each node of the delaunay mesh (black dot) is a voronoi cell's generating point.

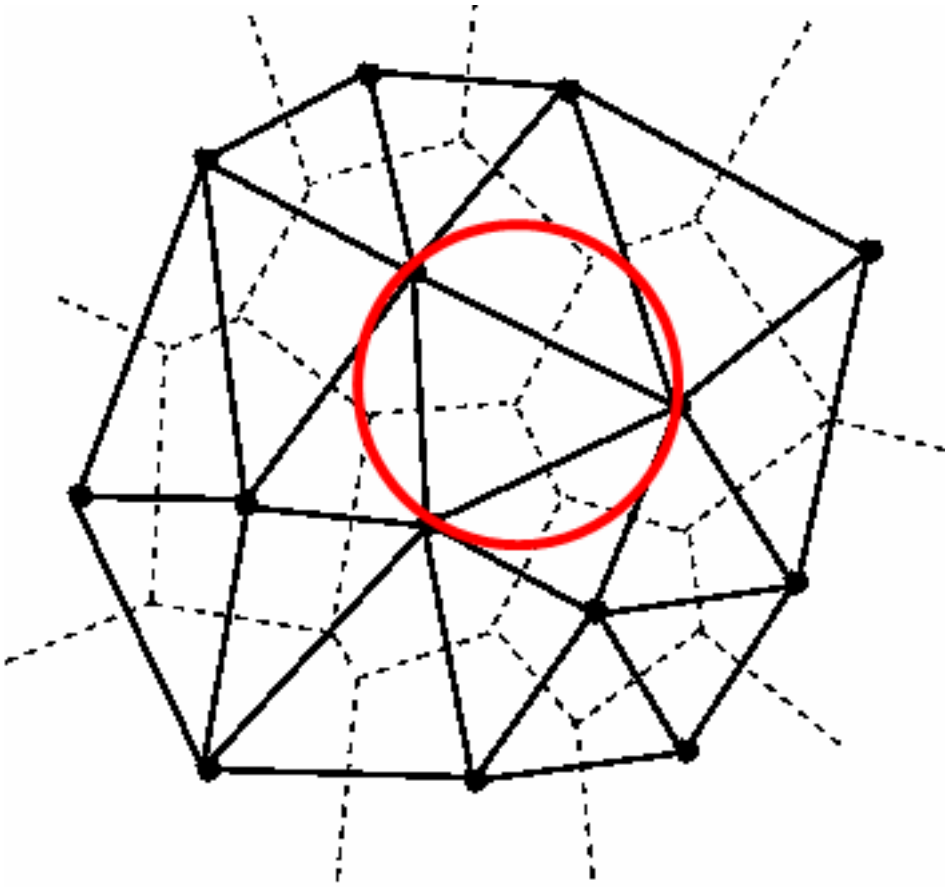

Figure 2.6: Dual mesh. Dashed lines are the voronoi cells, solid lines are the delaunay triangles. Red circle is a circumcircle [27]

The Delaunay triangulation is therefore optimal if the mesh vertices are fixed. In practice however, the positions of the vertices we want to optimize are unknown. The Centroidal Voronoi Tessellation (CVT), also known as the Lloyd algorithm, uses the dual property to produce an optimal mesh. The algorithm 1 iterates between 2 steps [28].

while *not optimal* **do**
    Determine the Voronoi diagram for a given set of points;
    Move the cell generating point to the center of gravity of the cell;
end

Algorithm 1: Steps of CVT algorithm

The result of the CVT iteration is shown in figure 2.7. It is shown to be isotropic (maximally uniform) Voronoi cells, thus producing points with a maximally uniform distribution within the domain. In 2D the dual Delaunay triangulation is also isotropic but in 3D getting the ideal triangulation is not so obvious. A plane can be completely covered without any gaps with equilateral triangles, but regular tetrahedrons cannot cover 3D space without gaps, therefore CVT often results in sliver tetrahedrons [29].

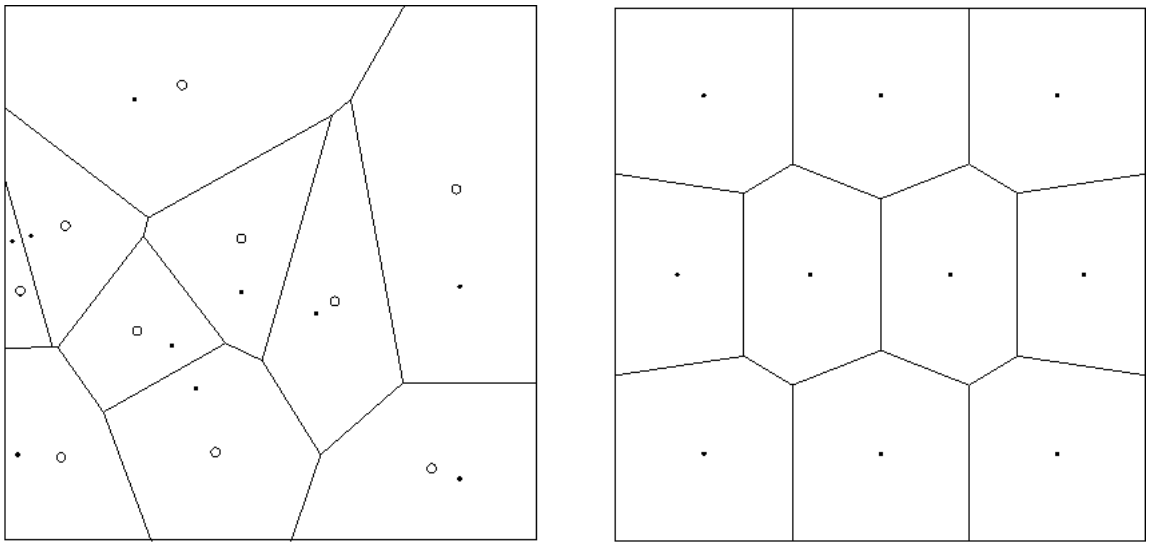

Figure 2.7: CVT algorithm. Black dots are the voronoi cell's generating points, circles are the center of gravity of the voronoi cells [28].

2.2.3 Optimal Delaunay Triangulation

CVT is a variational method [30] [28], vertex positions and connectivity updates are performed alternately to minimize the same quadratic energy eq. 2.12. This approach's consequence is that each step could be done optimally, and if the boundary is convex, then it converges.

$$E_{CVT} = \sum_{i=1..N} \int_{V_i} ||x - x_i||^2 \, dx \tag{2.12}$$

Where the x_i are vertex positions and V_i a local cell associated with each x_i.

Figure 2.8 shows that the minimization of E_{CVT} corresponds to minimizing the volume between a paraboloid and an underlaid circumscribing piecewise linear approximation [29]. But as mentioned before, the minimization of this energy optimizes the compactness of the dual Voronoi cells, but not the compactness of the simplices, therefore in 3D it produces slivers. Chen had the idea to use overlaid piecewise linear approximant (E_{ODT}) [31]. Applying this to eq. 2.12, results eq. 2.13. The integral is now taken over each 1-ring region Ω_i, therefore E_{ODT} energy is a quality of the mesh, not of its dual (Voronoi tessellation).

$$E_{ODT} = \frac{1}{n+1} \sum_{i=1..N} \int_{\Omega_i} ||x - x_i||^2 \, dx \tag{2.13}$$

This also gives a method for Delaunay triangulation, which can be seen on figure 2.9. The points are projected onto the paraboloid function's superficial, then if we project back to the plane the lower convex hull of the resulting point set we get the optimal delaunay triangulation of that point set.

But in real life applications we have a fix topology and we want to obtain the most accurate approximation of the function by moving the vertices of the mesh. Alliez et al. deduced that the local minimum (necessary) condition that the vertices are at the

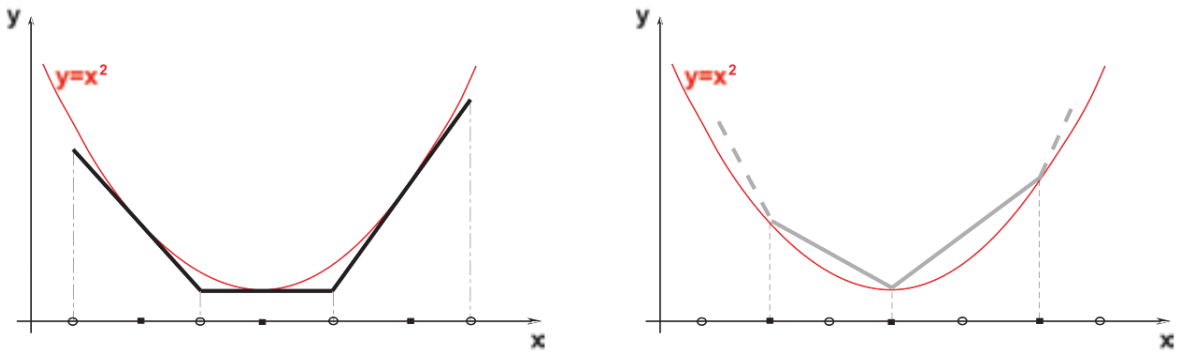

Figure 2.8: Approximating the same paraboloid function with an underlaid piecewise linear function (left side), and an overlaid piecewise linear function (right side) [29]

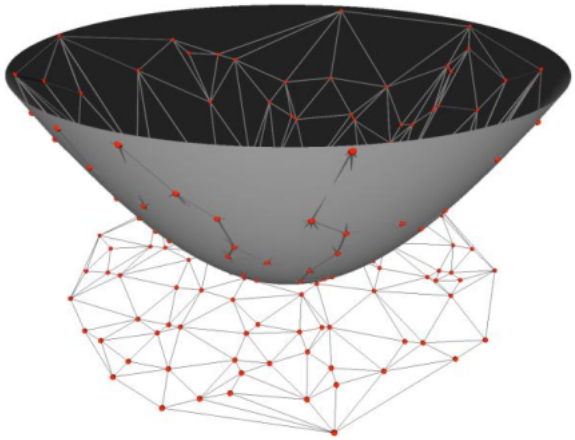

Figure 2.9: Gray is the paraboloid's superficial, red dots are the fixed vertices, grey lines are the delaunay triangulation's edges [32].

area-weighted average of the circumcircle's centers of the adjacent triangles. This means that the vertices must be repositioned according to eq. 2.14.

$$x_i^* = \frac{1}{|\Omega_i|} \sum_{T_j \in \Omega_i} |T_j| c_j \tag{2.14}$$

Where x_i is the new position of the i-th vertex, c_j is the circumcircle of the j-th adjacent triangle, $|T_j|$ is the area of the corresponding triangle, $|\Omega_i|$ is the area of the triangles adjacent to the vertex. Figure 2.10 demonstrates the variables in eq. 2.14. Algorithm 2 shows the steps of the Optimal Delaunay Triangulation (ODT), it iterates between two successive steps. The resulting triangulation is empirically found to consist of very uniform elements, free of sliver tetrahedrons and demonstrably minimizes the error of the discontinuously linear approximation.

while *not good enough* **do**
  Compute a Delaunay triangulation for a given set of points;
  Move the vertices according to eq. 2.14;
end

Algorithm 2: Steps of ODT algorithm

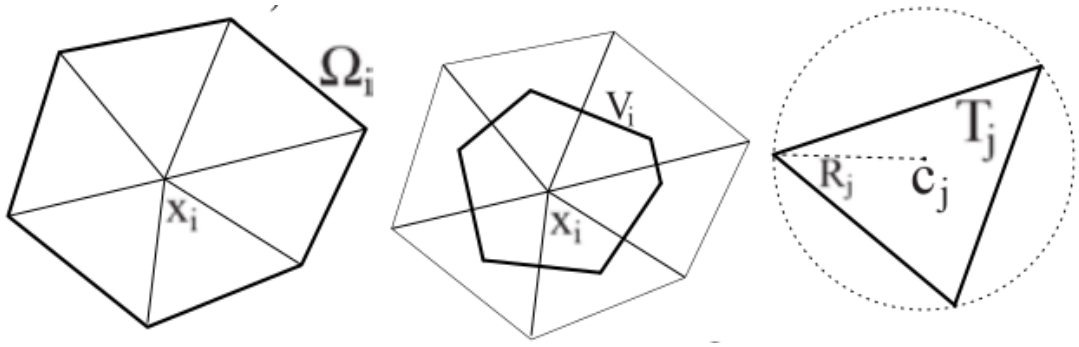

Figure 2.10: Caption

So far the ODT is able to produce good quality cells, but as mentioned before, being able to handle the mesh density and the boundaries is required to make a quality volume mesh. To achieve graded mesh Alliez et al. used a one-point approximation of the sizing field μ in a tetrahedron and defined the mass density as being $\frac{1}{\mu^3}$ since the local volume of a tetrahedron should be roughly the cube of the ideal edge size. In case of a triangle element the mass density is $\frac{1}{\mu^2}$.

$$x_i^* = \frac{1}{\sum_{T_k \in \Omega_i} \frac{|T_k|}{\mu^3(g_k)}} \sum_{T_j \in \Omega_i} \frac{|T_j|}{\mu^3(g_j)} c_j \tag{2.15}$$

where g_k is the centroid of T_k. The article shows a function what is ideal in terms of respecting the geometry, but theoretically any function can be used instead.

The treatment of the boundary complicates the method. The article uses quadrature points to preserve the discretization of the boundary. These quadrature points act as a pulling force and pull the points towards themselves, demonstrated on figure 2.11. For a convex boundary simple Delaunay triangulation is sufficient. In case of a concave boundary however, it must be taken into account that the Delaunay algorithm uses a convex hull of the triangulated points, the boundary to be preserved must therefore be added as a constraint and the outer part of the triangulation must be removed.

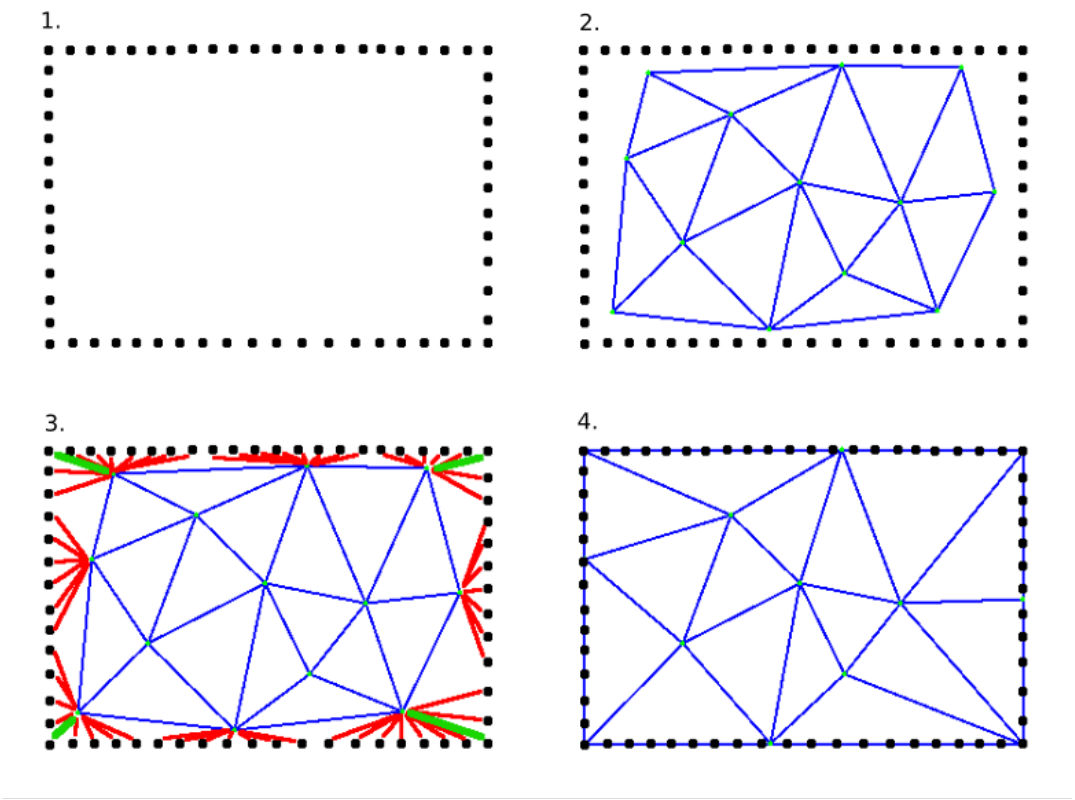

Figure 2.11: Quadrature points in action. Black dots are the quadrature points, blue points are the vertices of the mesh, purple lines are the mesh edges. Green and red lines are representing the "pulling force" of the quadrature points [33]

2.3 Artificial Neural Networks

Artificial neural networks (ANN) are models that try to solve certain problems by mimicking the way the human brain works. A neural network is a collection of interconnected neurons which defines a mapping $f : x \rightarrow y$ where x is the input variable and y is the output variable. The function f is s chain of many different functions, which can be represented through a network structure [34]. The mapping function's generale form as eq. 2.16, where x the input, y the output and Θ is the neural network's parameters. The process when the ANN tunes it's parameters to get the most accurate outputs is called learning or training.

$$\hat{y} = f(x; \Theta) \tag{2.16}$$

Based on the learning process, there are three groups of machine learning: supervised

learning, unsupervised learning and reinforcement learning. During supervised learning the training data is labeled, which means that we know the correct input-output pairs in advance. The model's performance is evaluated by how close is the predicted output to reality. The major drawback of this approach is that we need labeled data which is expensive to produce. To produce labeled data human work is required, which makes the process extremely expensive [35].

On the contrary unsupervised learning doesn't require labeled data, we don't know the correct output in advance, which makes the training process more cost effective. The network's main aim is to explore the underlying patterns and predict the output. Deep neural networks are capable of unsupervised learning.

The third type differs the most. The model (referred to as agent in reinforcement learning) learns how to react with an environment on its own. The agent can take actions which influence the environment's state, after each action the agent gets a reward. The purpose of reinforcement learning is to learn an optimal policy that maximizes reinforcement signal that accumulates from the immediate rewards, thus the agent must make a series of interrelated decisions without knowing every individual decision's correctness.

2.3.1 Structure

A neuron takes in an input vector and a bais as an input, then aggregates these values into a scalar value, then passes it into a nonlinear transformation function called activation function. The activation function is often denoted as ϕ eq. 2.17. The schematics of a neuron can be seen in figure 2.12.

$$y_i = \phi\left(b_i + \sum_{j=1}^{n} w_{ij}x_j\right) \tag{2.17}$$

where y_i is the output of the neuron, b_i is the bais, w_{ij} is the weight value between the neuron i and the previous layer's j-th neuron, these weights resemble the connections between the neurons and x_j is the j-th neuron in the previous layer [36].

The neurons get constructed to layers. The width of a layer describes how many neurons it has, the layers has three types;

- input layer
- hidden layer
- output layer.

A fully-connected, feed-forward network is shown in figure 2.13. It is the most basic type of neural network, they are also called as a multilayer perceptron (MLP). It's fully connected because each neuron is connected to all the neurons in the previous layer, it's feed forward because there are no feedback connections in which outputs of the model are fed back into itself.

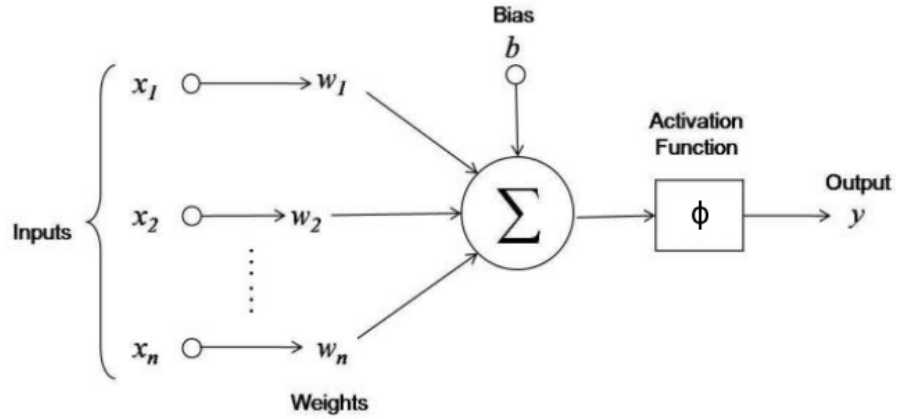

Figure 2.12: The in and outputs of a neuron [37].

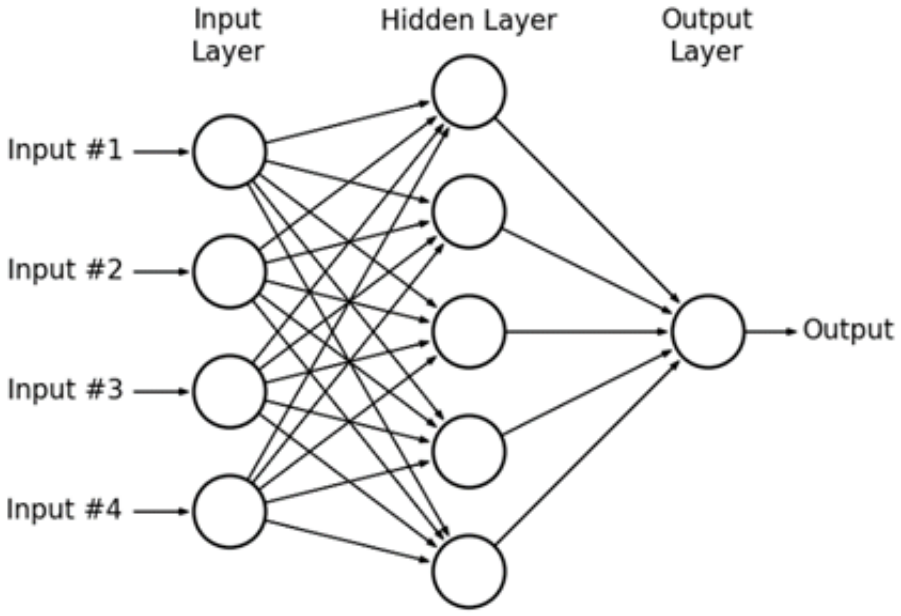

Figure 2.13: Structure of a feed forward neural network [38]

2.3.2 Activation functions

Activation functions must meet two criteria. First, if it receives correct inputs then the unit should activate and produce an output, otherwise it should stay inactive. Secondly, the activation must be nonlinear, otherwise the whole neural network becomes a simple linear function and won't converge.

The simplest activation function is the step function which gives an output of 1 if the input is positive and 0 otherwise. More often used function is the sigmoid. It is a good choice for simpler ANN implementations. Its drawback is that if there is a derivation during the training process the large outliers flatten out, thus useful features get lost. This phenomenon is called the vanishing gradient problem. For the same reason the binary step function can't be used for gradient methods [36].

To avoid the vanishing gradient problem, a rectified linear unit (ReLU) is used. ReLU gives an output of 0 if the input is a negative number, else it's output is the input. The three mentioned function are shown on figure 2.14 [39] [40].

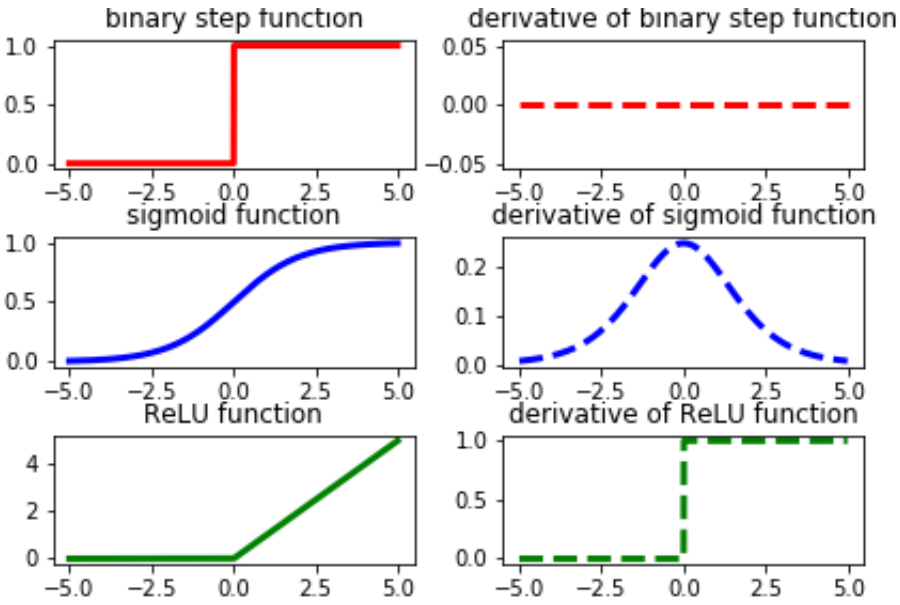

Figure 2.14: Activation functions and their derivatives

2.3.3 Deep learning

If there are multiple hidden layers in the network, then it's called deep neural network (DNN). Deep learning uses the backpropagation algorithm to optimize the weights and biases, thereby allowing hidden structures in the data. Some of the DNN's key features are [41]:

- They requires little human interactions
- The learning time scales linearly with the size of the training data
- Is capable of unsupervised learning

2.3.4 Backpropagation

The accuracy of a neural network is measured by the loss or error function. One common error function is the mean squared error 2.18.

$$E(x, \Theta) = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 \tag{2.18}$$

Where E is the error function in the function of the weights and biases and the inputs of the neural network, $\hat{y}_i$ is the output of the network and y_i is the expected output for the i-th input [9].

During learning, the value of this function must be minimized by varying the weights of the neurons. Since the error function is differentiable, gradient based methods such as gradient descent can be used to solve this optimization problem.

If we calculate the derivative of the error function with respect to the weights, and weights w_{ij} are changed in the opposite direction than this gradient, then it will be the direction of the fastest decrease. The update of the weight values are done via the delta rule 2.19.

$$\Delta w_{ij} = -\lambda \frac{\partial E(x, \Theta)}{\partial w_{ij}} \tag{2.19}$$

λ is the learning rate, which defines how big the change should be. If λ is big, then the changes in the weight is big, thus the loss function is moving faster to the minimum, but it can miss it and the loss starts to oscillate. If the learning rate is small the convergence slows down [42].

Since the error function only evaluates the accuracy of the output layer directly. The so-called backpropagation method is used to evaluate the error of all neurons in the network. To simplify Equation 2.17, let ν_j denote the aggregate of the j-th neuron , then eq. 2.17 can be written as 2.20.

$$\hat{y}_i(n) = \phi_i(\nu_i(n)) \tag{2.20}$$

Using the equation 2.20, the partial derivative of the error function w.r.t weight w_{ij}^k can be expressed using the chain rule as eq. 2.21:

$$\frac{\partial E(x, \Theta)}{\partial w_{ij}^k} = \frac{\partial E}{\partial \nu_j^k} \frac{\partial \nu_j^k}{\partial w_{ij}^k} \tag{2.21}$$

The first term is the local gradient shown in eq. 2.23. The second term is the partial derivative of the error function w.r.t weight w_{ij}^k, which is equal to the $i-th$ neuron's output in the k-th layer. Equation 2.21 can be reduced to eq. 2.22.

$$\frac{\partial E(x, \Theta)}{\partial w_{ij}^k} = \delta_j^k x_i^{k-1} \tag{2.22}$$

$$\delta_j^k = \frac{\partial E}{\partial \nu_j^k} \tag{2.23}$$

The local gradient can only be calculated directly at the output layer. To calculate the local gradients of the neurons in the hidden layers the chain rule has to be applied to 2.23, and utilizing the fact that the neurons outputs from the previous layer are the inputs of the next layer's neurons, it can be deduced the local gradient of the j-th neuron

in the k-th layer is equal to eq. 2.24, what is called the backpropagation formula.

$$\delta_j^k = \phi'(\nu_j^k) \sum_l^{r^{k+1}} \delta_l^{k+1} w_{ij}^{k+1} \tag{2.24}$$

Where l is the number of neurons in the next layer and r^{k+1} is the number of neurons in the k+1 layer. If eq. 2.23 is substituted back into eq. 2.21, then the partial derivative of the error function w.r.t a weight w_{ij}^k in the hidden layers can be calculated with eq. 2.25.

$$\frac{\partial E}{\partial w_{ij}^k} = \phi'(\nu_j^k) x_i^{k-1} \sum_l^{r^{k+1}} \delta_l^{k+1} w_{ij}^{k+1} \tag{2.25}$$

Each local gradient is dependent of the next layer's local gradients, therefore the error flows backward from the last layer to the first layer. On the contrary the input is passed forward, thus the name forward propagation. Since the error is also dependent of the neuron's output, first the forward propagation needs to be calculated in each iteration [43].

2.3.5 Hyperparameters

In addition to the model's weights and biases, the neural network has an additional set of parameters called hyperparameters. These external configurations refer to the following parameters:

- Number of hidden layer number
- Number of units in hidden layers
- Activation functions
- Learning rate
- Number of epochs
- Batch size
- Optimizers

Backpropagation updates the weights and biases of the neural network, but these values influences how optimally the neural network works.

The first four were discussed already. When the network goes trough all the inputs in the training data set is called an epoch. Batch size is the number of sub samples given to the network after which parameter update happens, so the epoch gets divided into smaller sets, and the learning algorithm iterates trough these sets. Optimizers are the method used to tune the network's weights and biases, they are the gradient based optimization methods mentioned.

2.4 Physics informed Neural Networks

The Universal Approximation Theorem states that an MLP with a hidden layer can estimate any function if the hidden layer has a sufficient number of neurons [44]. Hence, ANNs can be used to solve differential equations. Maziarraissi et al. have created a machine learning model for solving PDEs, the model is called physics informed neural networks (PINNs). This model also exploits another property of neural networks, the automatic differentiation schemes (e.g., backpropagation), therefore the dependence of the solution on the quality of the network can be eliminated [45].

PINNs consist of a simple FNN, the physical equations and boundary conditions are integrated into the network's error function. The output of the neural network will be used to estimate the solution of the PDE. Since $y(x)$ is unknown, $haty(x, \Theta)$ is learned using the PDEs. The error function is defined separately for the boundary conditions and for the interior domain, and a training dataset is also defined separately for each of the two error functions. Denote by T_f the residual points for the domain interior and T_b the residual points for the boundary conditions, then the loss function can be expressed as 2.26.

$$L(\Theta, T) = w_f L_f(\Theta, T_f) + w_b L_b(\Theta, T_b) \tag{2.26}$$

where

$$\begin{aligned} L_f(\Theta) &= \frac{1}{|T_f|} \sum_{x \in T_f} ||F(\hat{y,}x||^2 \\ L_b(\Theta) &= \frac{1}{|T_b|} \sum_{x \in T_b} ||B(\hat{y,}x||^2 \end{aligned} \tag{2.27}$$

$F(\hat{u,}x$ and $B(\hat{u,}x$ are solved by AD method such as backpropagation. The network is trained until the suitable Θ is find.

2.5 Machine learning aided meshing approaches

This section describes works where machine learning has been used to generate or optimize numerical meshes. There were papers made about these issues as early as the nineties. One approach is to optimize the density of the mesh using Kohonen maps also known as self-organizing maps (SOM) [46] [47]. SOMs are different from the ANNs described so far. The weights of the neurons in SOMs are describing the euclidean distance between each other. The inputs can be understood also as points inside the domain, and during the learning process only the the closest neuron's and it's surrounding neuron's weights are updated.

In this case Kohonen map's neurons are equivalent to the nodes of a numerical mesh. SOM will be the nodes of the mesh, and their distribution is therefore a probability distribution. The connections between the nodes must be defined before training, thus the training process only optimizes the existing nodes distribution. The inputs to the

network are points produced by a node density function. Since the edges of the elements in the mesh are predefined, in the case of irregular node distributions, skewed elements are formed as shown in figure 2.15. Another drawback of this approach is that since the teaching points are located inside the domain, no nodes will be on the edges, so the vertices on the boundaries must be defined by the user [48].

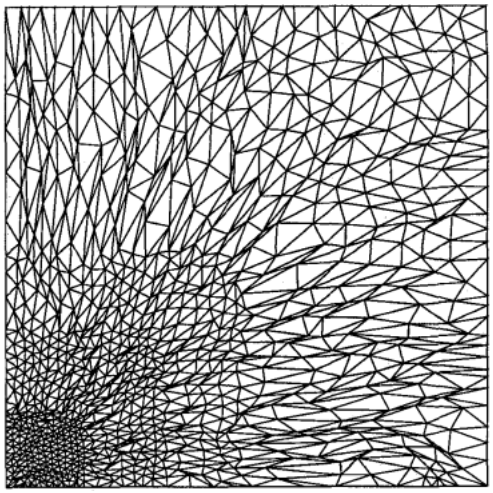

Figure 2.15: Unstructured mesh made with kohonen maps [48].

Solutions to the shortcomings of Kohonen maps are provided by Let-It-Grow (LIG) neural networks [49]. This model also starts from an initial mesh, but it is able to generate new nodes, therefore it can handle irregular node distributions. The user defines a probability density function (pdf) that describes the desired mesh density on the domain. The topological evolution of the network is divided into two phases. The first phase is similar to the SOM learning steps. For a given input x it finds the closest node, then it moves that node in the direction of x. In the second phase new nodes are placed where the pdf's value is low. After the number of the mesh vertices reaches the desired number, Delaunay triangulation is used to generate the mesh.

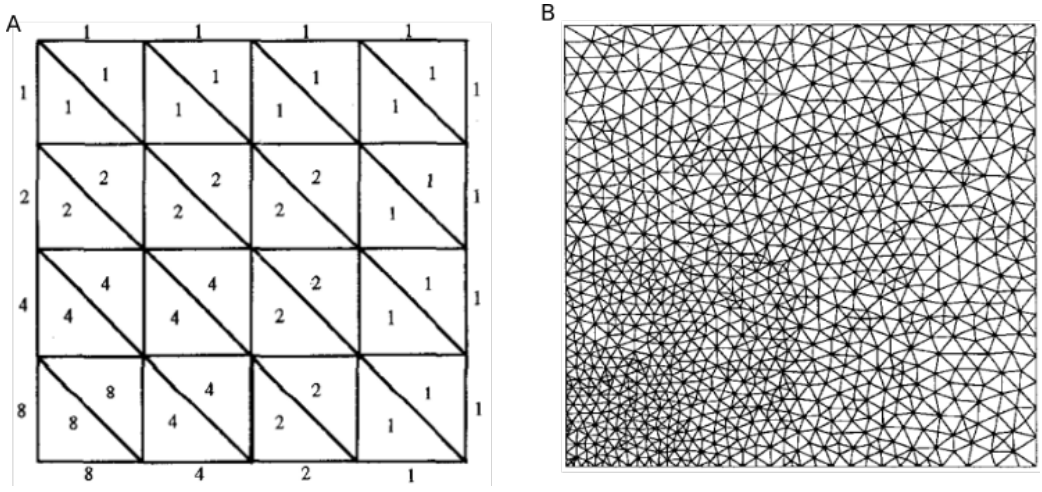

Figure 2.16: Mesh made by the let it grow network [49].

Simple FNN can be used as well to optimize mesh density. Chedid used single layer

MLP to predict the density of the mesh. A premade optimal mesh is constructed for a given domain, then this domain was divided into grids. Around these grid points the mesh density was determined. The input to the MLP was the coordinates of these grid points, and the output of the mesh was the predicted mesh density around the input point.

Cinar also used an MLP model; his approach was similar to Alfonzetti's algorithm. Cinar defined the boundaries of the domain with b-spline control points. These control points were the inputs of the MLP, and the outputs were nodes inside the domain. Then the distance between the b-spline control points is partitioned by an algorithm, and a meshing algorithm was executed recursively on the generated point set [50].

Unlike previous works, Yao has made an element extraction algorithm with a neural network. First points on the edges of the domain were defined. Then an algorithm feeds these points into the ANN in a specific way that the inputs contain the constraints of the domain. The mesh defines quad elements (cells) from these inputs. The element extraction is either done by placing a new point in the domain, or simply by defining a new edge between two existing vertices [51]. Figure 2.17 demonstrates possible ways of selecting input points. The algorithm iterates through the points on the boundary, and checks if they defines any edge corner between 35° and 135° with are adjacent boundary points. If such a reference point is found, then this point and its neighbours are be an input of the ANN.

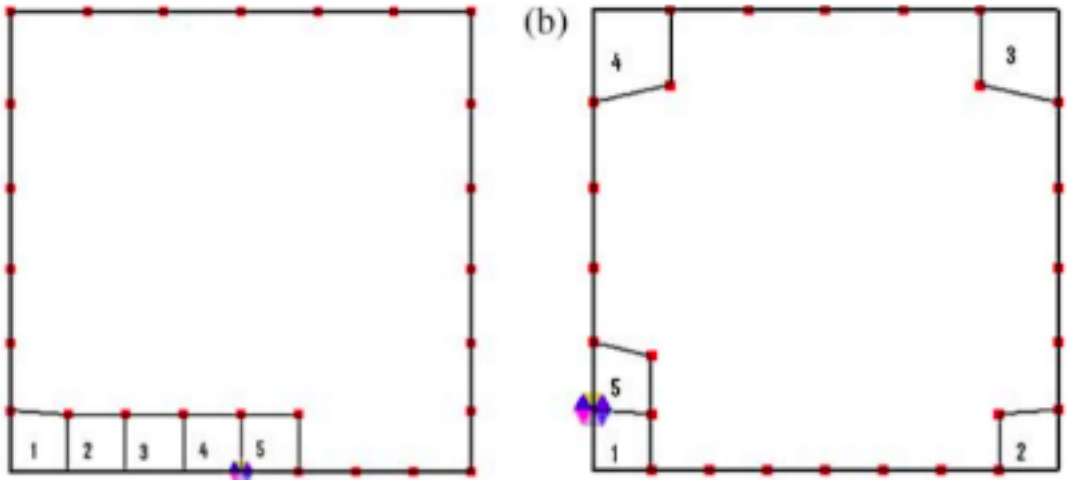

Figure 2.17: Selecting inputs for the ANN [51]

Zhang's MeshingNet is using a residual neural network architecture (ResNET). In these networks the information is able to skip certain layers, so from layer k the information can skip layer k+1 and can go directly to k+2 instead. The mesh density was predicted as a function of the geometry, the boundary conditions and physical properties. The training data was generated by running the same finite element simulation on a coarse mesh called LDUM and a dense mesh (HDUM). Then the error distribution was calculated from these simulations. The mesh density function produced by the ANN should be similar to this error distribution. The input to the ANN is the LDUM, and the output is the distribution of the mesh density over the domain as a whole. Then the mesh distribution was used to

refine the LDUM mesh using a meshing program [52].

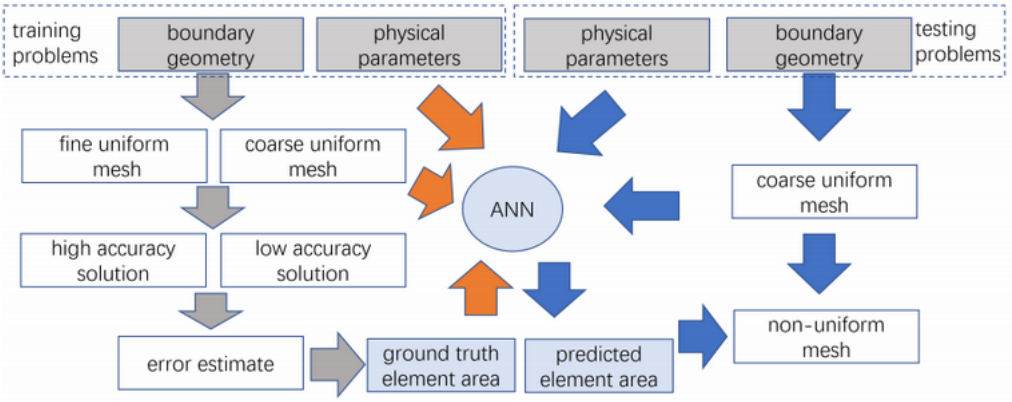

Figure 2.18: The structure of the MechingNet model. Grey arrows show the data generation pipeline, orange arrows represent the training process and the blue arrows represents the validation process. [52]

Chapter 3

Methods

3.1 OpenFOAM

OpenFOAM (FOAM stand for field operation and manipulation) is a free, open source software developed for solving continuum mechanincs problem, the dominant use case is computational fluid dynamics. It is basically a C++ library for making executable applications. These applications fall into two categories, solvers and utilities. Solvers are designed to solve computational problems, and utilities are mostly made for data manipulation. The structure of OpenFOAM can be seen on figure 3.1, it contains all the steps of a CFD simulation.

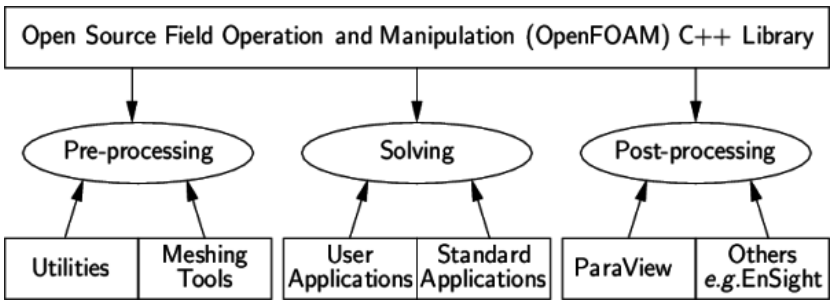

Figure 3.1: OpenFOAM's case file structure [53]

The program uses a case file structure as seen in figure 3.2. The *constant* directory contains the description of the mesh. OpenFOAM's mesh structure is called *polyMesh*, the program has utilities that enable the conversion of other mesh structures to *polyMesh* format. Physical properties are located here as well.

The *system* dictionary contains the parameters related to the numerical simulation. In *controlDict* the definition of the start and stop time, the time step and the output parameters happens. The *fvSolution* file defines the control parameters of the solver, *fvSchemes* contains the discretisation schemes. *0* folder contains the initial and the boundary conditions. During execution of the simulation each timestep gets its own time directory, which contains the values of the parameters in that timestep.

One unique feature of the program, is that it's only capable of simulating 3D problems,

thus to be able to compute 2D problems on a 2D domain the mesh has to be extruded. Extruding a mesh means that each point in the mesh get extruded in the plane to a unit distance in the direction of the normal of the plane. structures to *polyMesh* format.

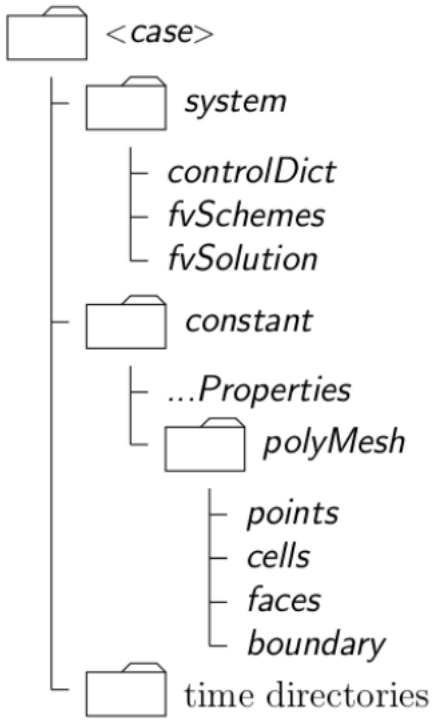

Figure 3.2: Lid driven cavity case domain [54]

3.1.1 Cavity case

This work used openFOAM's lid driven cavity case tutorial. It's geometry can be seen on figure 3.3. It's the simulation of an isothermal incompressible flow in a cavity that has a top boundary with a constant velocity. The domain is a square cavity with $d = 1m$ sides. To be able to run a simulation, the whole domain was extruded in the z direction by $0.1m$.

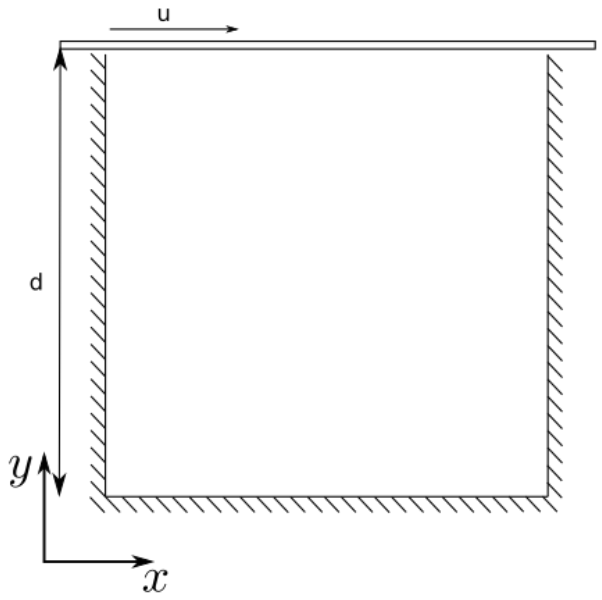

Figure 3.3: Caption

The meshing was done via *blockMesh* utility. This utility produces a structured grid with 400 (20x20x1) octahedron elements. Three types of boundaries were defined, a *movingWall*, *fixedWalls* and a *frontAndBack* boundary. *movingWall* defines the top of the cavity (the lid), *frontAndBack* defines the cavity's front and back face, which are set to type empty, thus the simulation will be executed as a 2D domain. The *fixedWalls* are the walls of the cavity with 0 velocity. The kinematic viscosity was set to $\nu = 0.1m^2/s$ in the *transportProperties*. The boundary and initial conditions were set for the pressure and for the velocity at 0\U and 0\p respectively. The boundary conditions (*boundaryField* variable) were the following:

- movingWall: for the velocity it was set to fixed value, for the pressure it was set to zerogradient
- fixedWalls: for the velocity it was set to noSlip, for the pressure it was set to zerogradient as well
- frontAndBack: it was set to empty for both parameter

The initial condition (*internalField* variable) of the pressure was set to uniform 0 since it is an incompressible case, so the kinematic pressure's absolute value is not relevant. The velocity's initial condition was set to uniform 0. The velocity's initial condition was also set to 0.

IcoFoam was the solver, it solves the incompressible laminar Navier-Stokes equations using the PISO (Pressure-Implicit with Splitting of Operators) algorithm. PISO is the extension of the SIMPLE algorithm. The starting time was set to $startTime = 0$, the end was set to $stopAt = 5$ and the time step was set to $deltaT = 0.005$.

The post processing tool used with OpenFOAM is paraViews which has a dedicated OpenFOAM release called paraFoam. The simulation results were extracted through this program then it was further processed with python scripts.

3.2 Python

Python is a high level interpreted programming language, it is widely used due to its easy syntax and its wide variety of available libraries. These libraries reduce the development time tremendously because most of the functions are already available, and ready to use. Another advantage of using libraries that they are often written in C/C++ and they are highly optimized, making their performance way better than if they were implemented in vanilla python. Such libraries were used in this work. Used in this work are: Pandas, NumPy, SciPy and matplotlib.

Pandas is used for data cleaning and structuring. It can handle a wide variety of data structures. In particular, it offers data structures and operations for manipulating numerical tables and time series [pandas]. It was used for processing the simulation data exported from paraFoam. NumPy is a library made for scientific computing. It is used for vector calculations. SciPy is also a scientific computing library, it has many submodules. SciPy spatial submodule was used for Delaunay triangulation here. Matplotlib is mainly a data visualization tool, it was used for making contour plots and for visualization of meshes.

3.2.1 ODT implementation

The meshing algorithm is implemented in 2D. Testing/debugging was performed on random point clouds. The initial state is a delaunay mesh created on the point cloud. This was generated with the scipy.spatial.Delaunay function.

```
1 from scipy.spatial import Delaunay
2
3 tri = Delaunay(vekt, qhull_options="QJ Pp")
```

This function takes a matrix of shape Nx2 for a 2D case as an input. The return value of the function is an object. The triangulation's data can be accessed by the simplices attribute. It contains the indices of the points, thus the coordinates can be by calling vekt[tri.simplices].

Updating the indices is done through a function called "updateVerts". It's pseudocode can be seen on algorithm 3. The inputs are the vertices, and simplices of the triangulation and the quadrature points. It Checks for each vertex if it's the closest neighbour of any quadrature points, if it is then their position will be updated accordingly, if not then the position will be updated according to 2.14.

Data: vertices, simplices of the Delaunay triangulation, quadrature points
Result: updated vertex positions
for *each vertex* **do**
  if *vertex is closest to any quadrature point* **then**
    place vertex in quadrature position;
  else
    continue;
  end
  for *each simplex* **do**
    if *simplex contains vertex* **then**
      calculate the area of the triangle what contains simplex;
      calculate the circumcircle center of the triangle containing simplex;
    else
      continue;
    end
  end
  update vertex's position according to eq. 2.15;
end

Algorithm 3: Pseudo code of updateVerts function. Data is the input and Results is the output of the function

$$
\begin{aligned}
D &= 2(a_x(b_y - c_y) + b_x(c_y - a_y) + c_x(a_y - b_y)) \\
U_x &= \frac{1}{D}\left[(a_x^2 + a_y^2)(b_y - c_y) + (b_x^2 + b_y^2)(c_y - a_y) + (c_x^2 + c_y^2)(a_y - b_y)\right] \\
U_y &= \frac{1}{D}\left[(a_x^2 + a_y^2)(c_x - b_x) + (b_x^2 + b_y^2)(a_x - c_x) + (c_x^2 + c_y^2)(b_x - a_x)\right]
\end{aligned}
\tag{3.1}
$$

For calculation the circumcircle centers equation 3.1 was implemented in calcTriMid function. Where a, b, c are the vertices of a simplex, x, y are the cartesian coordinates and $U = [U_x, U_y]$ is the centre of the circumcircle. The calculation of the triangle surfaces are done via the calcSurf function. Both functions usage presented below.

```
1 U = calcTriMid(vekt[simp])
2 T = calcSurf(vekt[simp])
```

As mentioned before the implemented algorithm has difficulties handling the boundaries. To prevent vertices getting outside of the boundary, control points called quadrature points were placed on the boundary of a domain. When updateVerts is called, first it calculates every quadrature point closest neighbouring vertex, this calculation is done via distBC function. A vertex and a quadrature point are neighbour only if the distance between them is the shortest for a given point cloud. The functions outputs are ids_arr stores how many quadrature neighbour has each vertex, $\frac{sum\_}{weights\_sum}$ is the weighted average of the neighbouring quadrature points position. It takes the vertices and the quadrature points as an input, since the weights of the points on the edges and on the corners are different, an extra array is needed where the control points in the corners are stored.

```
ids_arr, sum_, weights_sum = distBC(vekt[simp],quadratures_edge,quadratures_tip)
```

Data: vertices, quadrature points on the edges, quadrature points in the corners
Result: neighbouring relations, vertex positions
for *each quadrature point* **do**
  set closest distance to quadrature point to max;
  set the closest vertex's id to 0;
  if *quadrature in corner* **then**
    set weight to "infinity";
  else
    continue;
  end
  for *each vertex* **do**
    calculate the distance between quadrature and vertex;
    if *distance is the shortest* **then**
      update closest distance;
      update closest vertex id;
    else
      continue;
    end
  end
  store neighbouring connection;
  adjust vertex position and weight;
end

Algorithm 4: Pseudo code of distBC function. Data is the input and Results is the output of the function

The sizing field has two different implementation. First of the implementation is trough a function what calculates each vertex's distance from the domain's center of mass. The other implementation is when the updateVerts takes in an extra input variable what contains the weights of each vertex.

3.2.1.1 Geometry generation

Python scripts were used for generating different geometries to test the meshing algorithm implementation. A geometry consists of points that define the convex hull of the domain, and the quadrature points. After the definition of the domain random points were generated inside the boundaries. Scripts were made for generating with circles different radius, rectangles and convex polygons.

To make a circle, the radius of the circle, and how many vertices are desired inside the boundary have to be defined. Polar coordinates as shown in eq. 3.2 were used for generating points inside a circle with makePoints function. The function returns a numberOfpoints by 2 shaped array.

```
vekt = makePoints(radius, numberOfPoints)
```

$$\begin{aligned} \alpha &= 2r\pi K \\ p_x &= r\cos\alpha \\ p_y &= r\sin\alpha \end{aligned} \tag{3.2}$$

Where α is the polar angle, r is the radius, K is a random number between 0 and 1, p_x and p_y is the Cartesian coordinates of point p. If $K = 1$ then the script generates quadrature points.

The same function was used to generate random convex polygons. If we generate n number of random points, and we take their convex hull, then it will results a polygon which has sides between 3 and n. This approach is easy to implement but its disadvantage is that as $n \to \infty$ the polygon will converge to a circle.

To make a polygon, first the makePoints function gets called, then these points get passed to ConvexHull from scipy.spatial package which then returns the convex hull of the random point cloud, what defines a convex polygon.

```
from scipy.spatial import ConvexHull
import numpy as np

vekt = makePoints(1, numberOfPoints)
hull = ConvexHull(points)
polygon = points[hull.vertices]
```

Randomly distributed vertices inside the polygon can be achieved with makePoints. The convex hull of the polygon defines the domain, matplotlib's path can be used to check if a point is inside a domain or not. Path is an object, and it's method contains_points takes in an array of points, and returns a boolean if the array of points are inside or outside of the object.

```
from matplotlib import path

domain = path.Path(polygon)
domain.contains_points(v)
```

MakePoints was executed til the number of vertices inside the domain reached the desired amount as shown in algorithm 5. Quadrature points then were placed on each edge with numpy's linspace function.

```
import numpy as np

k = points_inside_polygon[hull.simplices]
```

```
quadrature_points = np.column_stack((np.linspace(k[0][0][0],k[0][1][0],n_bc),
                        np.linspace(k[0][0][1],k[0][1][1],n_bc)))
```

Where k is a 3D array, the 1D contains the edge defined by two simplices, the 2D defines the two simplices and the 3D defines the Cartesian coordinates of the simplices. Linspace divide the edge into n_bc number of equidistant points as shown below. These steps were executed for each edge. Same method was used to generate the quadrature points of rectangles.

while *number of points inside the domain is satisfied* **do**
  generate a point with makePoints;
  if *points inside domain* **then**
    add point to vertices;
  else
    continue;
  end
end

Algorithm 5: How random points inside a polygon are made

To make a rectangle a function was made called makeRectangle. It's inputs are the length of the edges and the number of vertices wanted inside the domain, the outputs are two arrays, one contains the vertices and the other contains the corners. The generation was done with eq. 3.3.

$$\begin{aligned} dist &= a - \frac{a}{100} \\ p_x &= distK \\ p_y &= distJ \end{aligned} \tag{3.3}$$

Where *dist* is the edge length's 99%, K and J are random numbers between 0 and 1, p is the vertex position with Cartesian coordinates x, y.

Sizing field got implemented in two ways, one is weighted the vertices in the function of the distance from the domain's barycenter, the other method is used the weights produced with and ANN, what will be explained in the upcoming sections. Each circle generated has a barycenter $[0, 0]$, and each rectangle's barycenter can be calculated as $\left[\frac{a}{2}, \frac{a}{2}\right]$, where a is the edge's length. For the polygons the barycenter of the boundary was calculated, which is the weighted average of the edges. An edge of a polygon is defined by two points, and end point p_1 and a start point p_0.

Data: vertices of the polygon's boundary
Result: barycenter coordinates
for *each edge* **do**
  calculate edge length;
  calculate the edge's mid point;
end
calculate the weighted average of the mid points;

Algorithm 6: Pseudo code of calculating a polygons barycenter

Gmsh was used for the mesh extrusion, it is an opensource finite element meshing software. Gmsh mesh structure is .msh. With python's meshio package the generated vertices were saved to a .msh file. To convert the generated mesh to .msh format the points and the cells had to be defined for the writer first. The points is the array which contains the vertices of the mesh. The cells are the simplices of the object returned by the Delaunay function, the type of the cells also have to be given to the function.

```
import meshio

cells = {"triangle": tri.simplices}
meshio.write_points_cells(
        "teszt.msh",
        points,
        cells,)
```

Then the mesh was imported to gmsh where the extrusion was executed. The extruded mesh then was exported to a .vtk file.

3.2.2 Deepxde

Deepxde is a deeplearning framework built on top of google's TensorFlow, what is a symbolic math library written in python and C++. It can run on multiple computers, CPUs simultaneously. It can also utilize GPU architectures. Deepxde was used to solve the Navier-Stokes equations with a physics informed neural network on the lid driven cavity domain.

3.2.2.1 Cavity model

To solve the Navier-Stokes equations with DeepXDE the following steps are required:

1. Specify the computational domain
2. Define the NS equation
3. Set the boundary and the initial conditions
4. Generate training data
5. Compile the model and set the hyperparameters
6. Train the model

DeepXDE's flow around a cylinder sample case was modified for solving the cavity case.

Step one is done trough the framework's geometry module. We want to solve the lid driven cavity case, thus our domain is a 1 by 1 rectangle, which can be defined as:

```
from deepxde.geometry.geometry_2d import Rectangle

geom = Rectangle([0.0, 0.0], [1.0, 1.0])
```

Then each boundary got defined with a function with two inputs, a boolean and a coordinate point. Numpy's isclose function checks if the point is on the boundary, then it set's the boolean's value accordingly. The definition of the bottom wall boundary can be seen below. The function check if point x's y directional component is 0. The rest of the boundaries were defined similarly.

```
def wall_bottom_boundary(x, on_boundary):
    return on_boundary and np.isclose(x[1], 0.0)
```

The gradients of the pressure and of the velocity is calculated with TensorFlow's gradients function, using these gradients equations 2.5 and 2.6 was implemented. The initial and boundary conditions were implemented with DeepXDE's DirichletBC class. It's inputs are the geometry, the boundaries and the a function stating the exact solution of the boundary. The output is the components values that satisfy the boundary condition.

```
from deepxde.boundary_conditions import DirichletBC

def zero_velocity(x):
    return np.zeros((x.shape[0], 1))
wall_b_x = DirichletBC(geom, zero_velocity, wall_bottom_boundary,
                       component=0)
```

Next the training data was generated with the PDE class, it takes the previously defined objects, and the desired amount of residual points on the boundary and inside the domain as inputs. Output is randomly sampled residual points on the boundary and inside the domain.

```
from deepxde.data.pde import PDE

data = PDE(geom, navier_stokes,
           [inlet_x, inlet_y, wall_l_x, wall_l_y, wall_r_x,
            wall_r_y,wall_b_x,wall_b_y],
            num_domain=1000,num_boundary=1000, num_test=100
            )
```

num_domain defines the residual points inside the domain and num_boundary defines the residual points on the boundaries.

For the neural network a feed forward neural network was used. The network's hyperparameters were set as shown below:

```
from deepxde.maps.fnn import FNN
from deepxde.model import Model

layer_size = [2] + 5*[20] + [3]
activation = "tanh"
initial = "Glorot uniform"
optimizer = "adam"
learning_rate = 0.001
net = FNN(layer_size, activation, initial)
model = Model(data, net)
model.compile(optimizer, lr=learning_rate)
```

The network has an input layer with 2 neurons, 5 hidden layers with 20 neurons each and an output layer with 3 neurons. Every neuron is a tanh neuron, which means their activation function is tanh. To train the model model.train() method has to be called. Early stopping stops the learning if the convergence is achieved, epochs is the number of iteration trough every data points.

```
early_stopping = EarlyStopping(min_delta=1e-8, patience=4000)
model.train(epochs=10000, display_every=1000, callbacks=[early_stopping],
            disregard_previous_best=True)
```

3.3 CTGAN

CTGAN is a Deep Learning based Synthetic Data Generators for single table data. The framework has two implemented models CTGAN and TVAE. A script was made for generating training data for the CTGAN model. The ODT implementation was used for this purpose as well. The script saved the data into two csv files.

The script was making random circles with radius between 1 and 2 and 30 quadrature points. Then generated 30 randomly placed vertices inside the domain, and the ODT algorithm was executed on the vertices 30 times. The initial positions of the vertices was then saved into test_in.csv file, and the optimized positions were saved to test_out.csv file.

In the csv files each row was a unique case, and the vertex positions were the columns table 3.1 demonstrates the csv file's structure. The csv files then were imported to a

	p_x^1	p_y^1	p_x^2	p_y^2	...	p_x^{30}	p_y^{30}
mesh1							
mesh2							

Table 3.1: Table demonstatring the training data csv file structure

pandas data frame and the vertex coordinates type was set to float16 to keep the memory

usage to the minimum. The dataframe variable name was data. Initialising the neural network was very simple since it just had to be imported from the package.

```
from ctgan import CTGANSynthesizer

ctgan = CTGANSynthesizer(batch_size=2)
ctgan.fit(data, df.columns.to_list(), epochs = 1)
```

The model's fit method starts the training process.

3.3.1 Google colaboratory

For training the machine learning models Google's free cloud computing platform called colaboratory was used. Colab is a hosted Jupyter notebook service that requires no setup to use, while providing free access to computing resources including GPUs [colab]. It provides 13GB of memory and 70GB of hard disk space. Users can also train models on a GPU but this function is limited to 12hours for each user per day. Any python package can be installed during a session, it's also possible to run unix shell commands in it.

3.4 Post processing

As mentioned before the simulation results were interpreted via python scripts as well. The results was extracted to a .csv file trough paraFoam. For the visualization of the velocity magnitude matplotlib's contour plot utility was used, since it requieres a meshgrid as input the csv file had to be converted to a meshgrid data structure. Since the simulation results were plotted at the vertices of the mesh, the coordinates could be generated with numpy.

```
import numpy as np

x = np.linspace(0,1,21)
y = np.linspace(0,1,21)

xx,yy = np.meshgri(x,y)
```

The result will be the same grid as the mesh used for the cfd simulation, xx array contains all the x coordinates and yy array contains all the y coordinates. Now the simulation results have to be converted to an x by y array as well. Luckily numpy's argsort function can do the job. First the csv file was imported to pandas dataframe, then the rows where the z directional coordinate was not 0 were deleted. After that the velocity's x and y components were used to calculate the velocity magnitude, then a numpy array was defined containing the coordinates and the velocity magnitude.

```
sortedby1 = simresults[simresults[:,1].argsort()]
zz = np.zeros(gx.shape)
for idi,i in enumerate(np.linspace(0,1,21)):
    xsort = sortedby1[np.where(sortedby1[:,1]==round(i,2))]
    xsort = xsort[xsort[:,0].argsort()]
    zz=np.vstack((zz,xsort[:,2]))
print(zz)
zz = np.delete(zz,0,0)
```

the simresults contains the coordinates and the velocity magnitude. Then the simulation results were sorted by the y coordinates first, then in a loop the sortedby1 array was sorted for each x values it was sorted.Then it was added to array zz. To access the values of the velocity at x=i and y=j we have to call zz[i][j], this is how contour plot plots the values.

```
import matplotlib.pyplot as plt

plt.contour(xx, yy, zz, colors='k', linewidths=0.2, levels=50)
```

For vectorfiles matplotlibs quiver function can be used. For quiver the x and y directional velocity components have to be given separatedly.

```
plt.quiver(xx,yy,up,vp)
```

where up is the x directional velocity component and vp is the y directional component.

If the velocity magnitude is predicted at a vertex, the following function was used to convert the velocity into a weight.

Data: velocity, vertices
Result: weights
for *each vertex* **do**
  if *velocity at vertex* **then**
    set the weight of the vertex in the function of the velocity;
    update closest vertex id;
  else
    set to a low value;
  end
end

Algorithm 7: Pseudo code of ordering weights from PINN to vertices

Chapter 4

Results and discussion

4.1 CFD simulation results

The lid driven cavity case is one of the simplest simulation of OpenFOAM, which is why it was chosen. It runs fast and gives accurate results, therefore it's perfect for evaluating the performance of the machine learning models, and testing out the requirements of a mesh to be compatible with the OpenFOAM software.

The mesh was generated by the blockMesh utility, it produced 400 quad cells as shown in figure 4.1. The values calculated in the nodes of these mesh will be used to evaluate the accuracy of the PINN model. The simulation converged around the 8th second as shown in figure 4.2. The execution time was no longer than a few seconds. The lid velocity was set to $1m/s$, the results of the simulations are shown in figure 4.3. The degree of deviation from these results becomes the measure of the accuracy of the PINN model, the smaller the deviation the better the model's accuracy.

Translated with www.DeepL.com/Translator (free version)

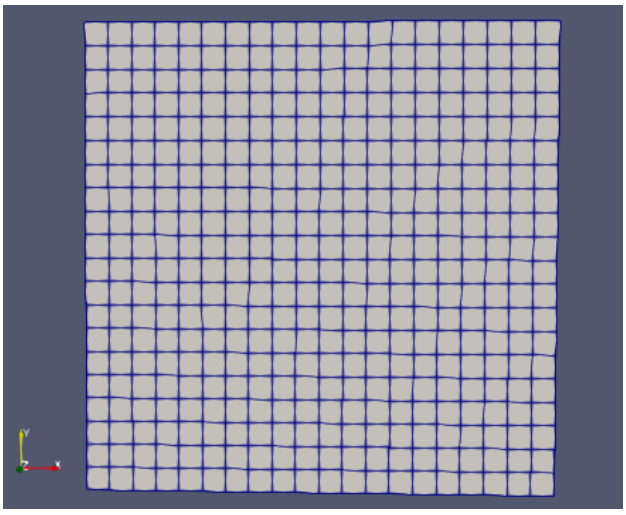

Figure 4.1: blockMesh dict generated 400 cells in a square with unit lenght edges

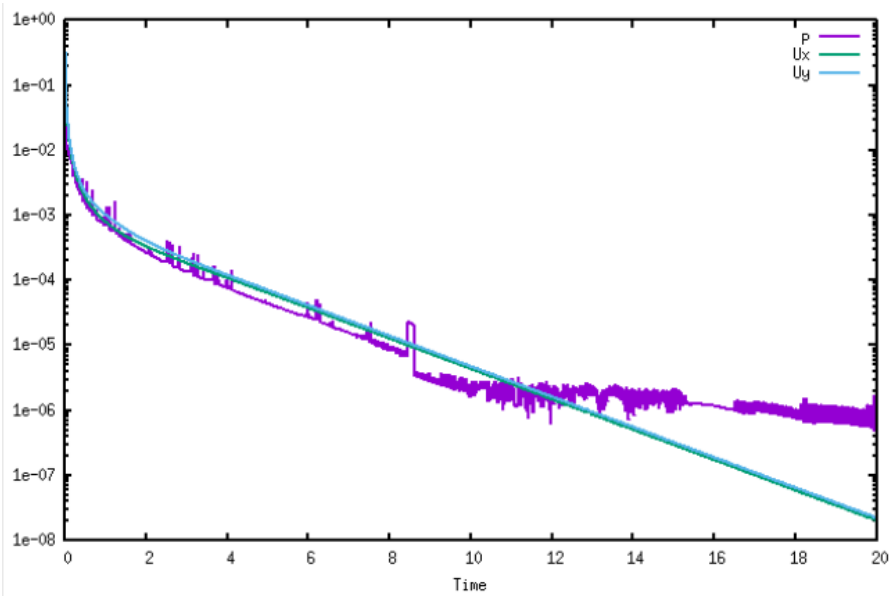

Figure 4.2: Residuals of the simulations of 100Re flow

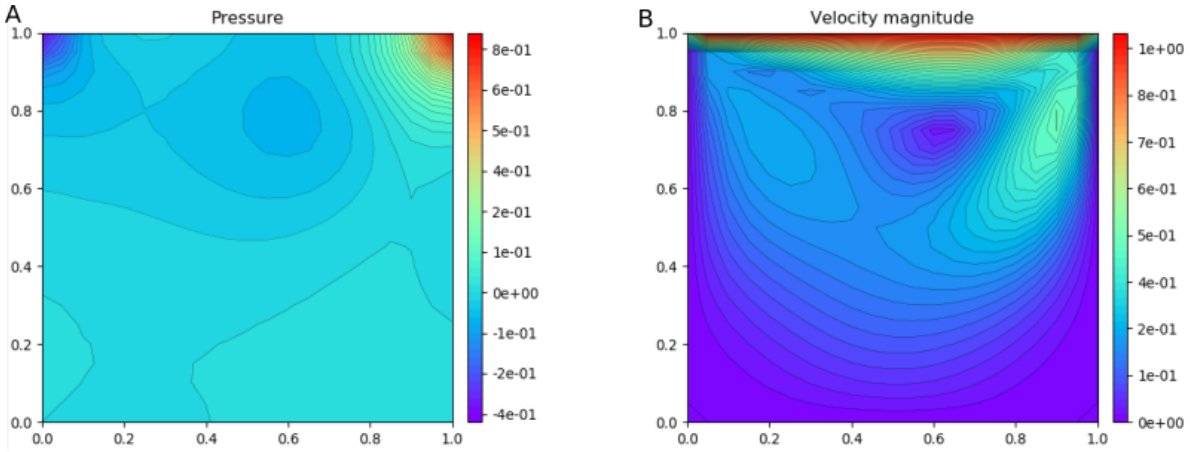

Figure 4.3: Pressure (fig. A) and velocity magnitude (fig. B) of the 100Re flow in the lid driven cavity

4.2 PINNs results

The loss function is already implemented in DeepXDE framework, but the hyperparameters can be changed. The hyperparameters were tuned manually the following way:

1. Set different network sizes and select the most accurate and the fastest one with a relatively good accuracy. The learning rate was set to 0.001 and the batch size were set to 16.

2. Then on these two models the batch size were changed from 16 to 32 then to 64

3. After finding the best batch size the learning rate was changed

The number of epochs were set to 80000, and didn't got changed. The activation function for each neuron was hyperbolic tangent. Since there are 2nd derivatives in the NS, ReLu couldn't be applied. The other option could have been sigmoid function as well.

It's important to note, that the speed of convergence can differ since the initial weights are set randomly at the beginning of each model training. The intitial weights could have been set to fixed values with defining the seed() of the tensorflow's random number generator.

The results of the first step can be seen in table 4.1. The layer size is interpreted the following way; inside the square brackets the number of neurons are defined, the first bracket is the input layer, the last one is the output layer, and the layers in between are the hidden layers. The "Best epoch" column shows at which epoch did the network performed the best, if the next 4000 epochs didn't yield better results, then the best state so far were saved, and the training stopped.

The learning time shows how much time it took the network to find the optimal weights and biases. If one of the model reached 80000 it means it's loss function did not converge yet, thus even if it's the fastest model it got eliminated. Models with maximum 0.1 max error are considered as well. Max error shows the maximum deviation from the CFD simulation's velocity magnitude results.

According to these criterias the models highlighted with green in table 4.1 got selected. The error of the pressure field was not taken into consideration when evaluating the accuracy, since as long as it converges the different magnitudes can be justified [maziarraissi]. The first model in table 4.1 was the most accurate, the other selected model was the

Layer architecture	Best epoch	Learning time[sec]	Max error[m/s]
[2]+[32]+[16]+[16]+[32]+[3]	55000	490	0.09
[2]+[16]+[8]+[16]+[3]	80000	625	0.4
[2]+2[50]+[3]*	80000	470	0.3
[2]+[300]+[3]	80000	600	0.4
[2]+5[20]+[3]*	61000	572	0.1
[2]+8[20]+[3]*	28000	382	0.2
[2]+4[50]+[3]*	29000	300	0.1
[2]+8[50]+[3]*	25000	230	0.4

Table 4.1: The results of the training of different network architectures. The green rows are the best performing ones.

second to last because it had the 2nd best accuracy with the fastest learning time. So far the batch size was kept on 16, in phase 2 it got changed to 32 first then 64. The results with different batch sizes can be seen in table 4.2.

Layer architecture	batch Size = 32			batch size = 64		
	B.Ep.	Time	MaxEr.	B.Ep	Time	MaxEr.
[2]+[32]+[16]+[16]+[32]+[3]	58000	530sec	0.1	63000	600sec	0.1
[2]+4[50]+[3]*	22000	240sec	0.08	26000	280sec	0.1

Table 4.2: The effect of the batch size on the learning. B.Ep. = best epoch, MaxEr.= max error, and time is the learning time

The $[2] + 4 * [50] + [3]$ network with a batch size of 32 was the most accurate model, the increased batch size made a significant difference. Next the learning rate of this combination was tuned as seen in table 4.3. It was observed that decreasing the learning rate to 0.0005 yielded the best results. If it was decreased further the convergence got too slow, if it got increased then the model got stuck in a local minimum. The highlighted values are produced with 0 set as seed to tensorflow's random number generator.

	Max error	Best epoch	Time
lr=0.0007	0.1	24000	280sec
lr=0.0005	0.08	39000	470sec
lr=0.0003	0.1	74000	707sec
lr=0.005	0.2	13000	95sec

Table 4.3: The effect of different learning rates. lr = learning rate.

The loss history of the highlighted model in table 4.3 is shown in figure 4.5, and the best prediction on the testing data set is shown on figure 4.4. The predicted flow field got compared to the CFD result as well, it is shown in figure 4.6. As mentioned before the nodes produced with blockMesh's were used as reference points, thus these were the inputs of the PINN model. The model then predicted the flow field, and taking the difference between the simulated and the predicted values results in the error contour plots.

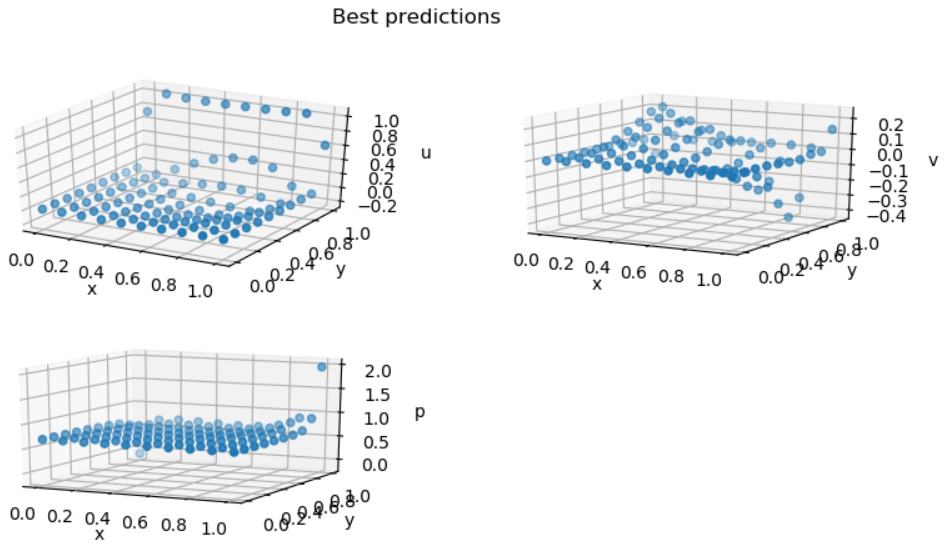

Figure 4.4: Networks each output's value on the validation set. The x and y axis are the x and y coordinates respectively, and the third axis are the outputs of the model. Top left corner is the velocity's x directional component, top right corner is the velocity's y directional component. The pressure is plotted on the bottom figure.

Figure 4.5: The loss history of the best performing hyper parameter set. Training loss shows the loss compared to the training data, and the validation loss shows the loss compared to the validation data.

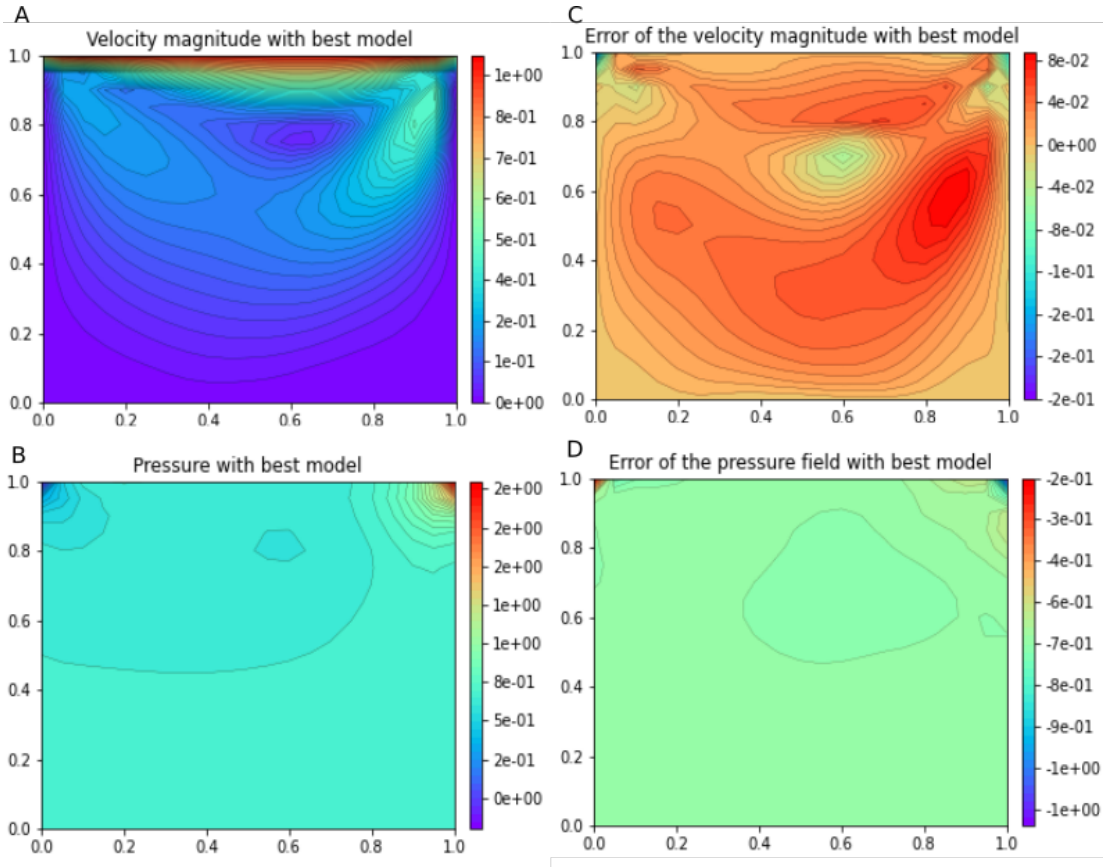

Figure 4.6: Contour plots of the flow fields and their errors. Fig. A shows the velocity magnitude, and fig. C is the difference from the CFD results. Fig. B is the pressure field and fig. D is its difference from the CFD results.

4.3 Meshing algorithm

The Optimal Delaunay triangulation (or Variational tetrahedral meshing) algorithms were selected because the experience gained from the surface meshes can be very easily transferred to volume meshes. Since in 2D the areas of triangles and the center of circles can be written around have to be calculated, in 3D it is enough to just rewrite the functions by calculating the volumes of tetrahedra and the spheres and the center of which can be written around.

During the development of the python script tests the debugging was done on circle geometries. The program was tested with and without boundary handling, and with and without sizing fields. First the vertex position update function was implemented. In this stage the boundary was defined by 9 fixed nodes on the circumference of the circle. It was then observed that after a vertex came out of the domain, it began to converge to infinity. It was caused by the fact that once the vertex is outside of the domain, it is part of only one simplex, and that simplex has two fixed vertices. On the other hand, the l-ring of the fixed points also included other triangles, so after a while they also "exploded". Figure B of the figure **??** shows this phenomenon.

If 100 quadrature points were placed on the circumference, this problem does not occur as shown in figure **??** C. It can be seen in pseudocode **??**, it contains two for loops, so the runtime increases exponentially with the number of quadrature points. Runtime could be improved with more complex data structures such as kd-trees.

4.3.1 Sizing field

For the sizing field, the distance from the domain's center of mass was used first. The result of this approach is shown in Figure **??** Figure D. This function produces denser mesh around the edges, and big triangles around the center, the transition between the two is getting less smooth as the number of iterations increase.

The program was also tested on convex polygons. Figure **??** illustrates the effect of the number of quadrature points. If not enough quadrature points were defined on the edges (**??** B), vertices can get outside the boundary, but have to keep in mind that the runtime increases exponentially with the number of quadrature points on the boundary, and after a certain number these points have no effect on the mesh quality (**??** D).

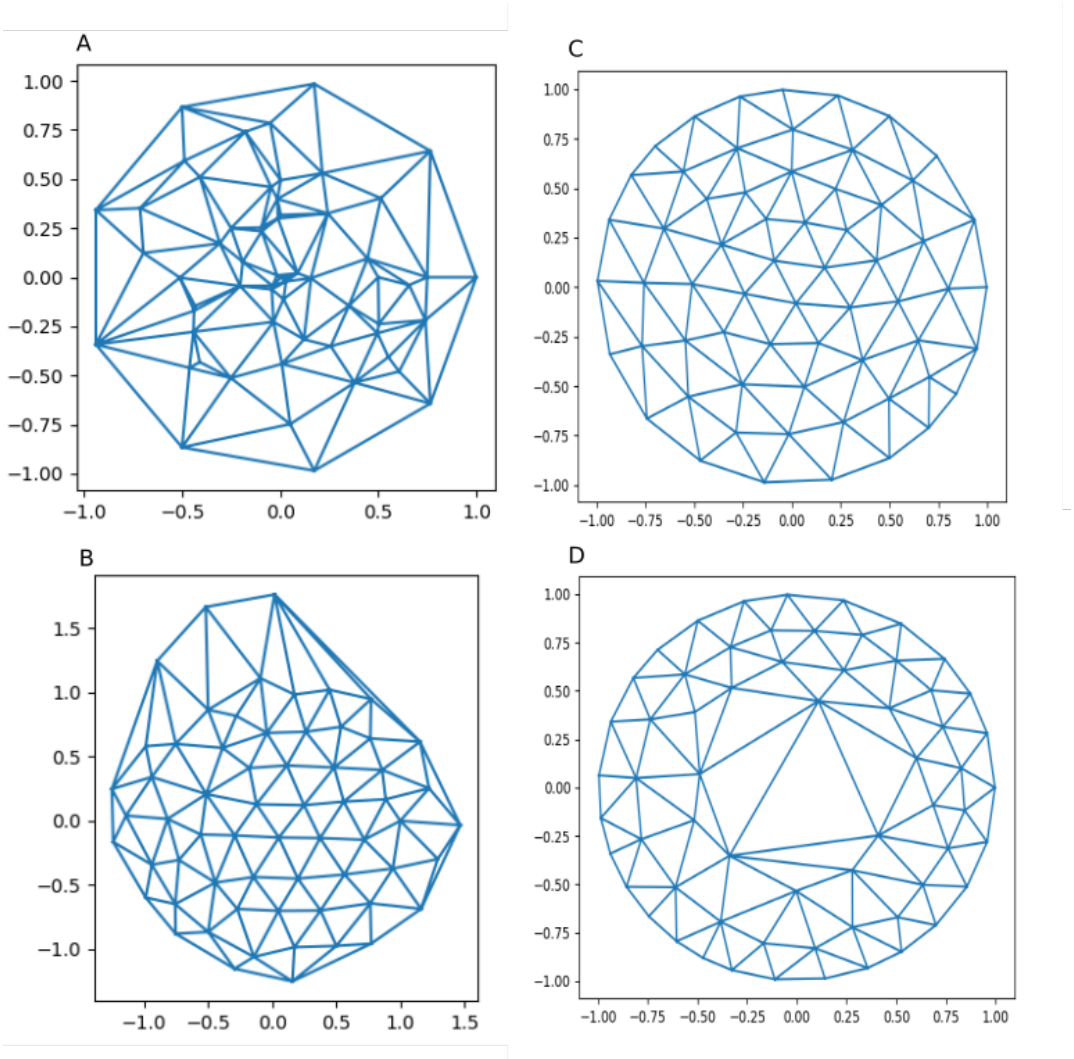

Figure 4.7: Meshes on the same point cloud with different ODT implementation. The initial Delaunay mesh can be seen on fig. A, it has 50 randomly placed vertices in a 0.8 radius. Fig. B shows the Delaunay mesh after 35 iterations with out quadrature points. Fig C. shows the mesh after 35 iterations with 100 quadrature points. Fig. C shows the mesh with sizing field in the function of the distance from the centre of the circle.

4.3.2 PINNs as sizing field

The idea behind using the flow field predicted by PINN is that once the model is trained it can predict any value of the flow at any point of the domain in the fraction of a second on a laptop. And the speed of prediction is not dependant on the complexity of the problem. So far CFD could be replaced by PINNs, the accuracy of a CFD simulation is still better, so if high precision is required then using PINNs for mesh grading could be a viable option.

Two approaches were tested for utilizing the predicted flow field for mesh grading:

- Place down the highest velocity points as new vertices
- Use the points with the highest velocity magnitude as quadrature points

The geometry was defined with 100 quadrature points on each edge of the rectangle, then

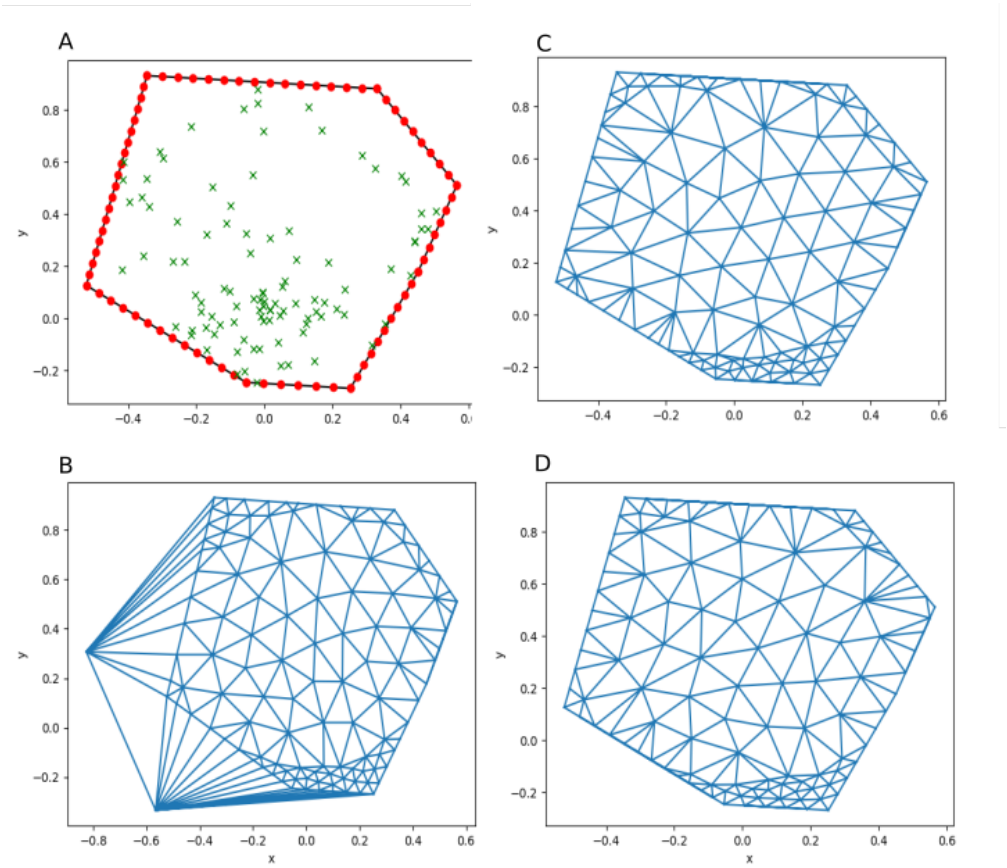

Figure 4.8: Delaunay mesh optimization on the same convex polygon. 50 iterations were made on 50 vertices, on figure C with 20 quadrature points, on figure B with 100 quadrature points and on figure D with 1000 quadrature points. Figure A shows the initial point cloud with green x-es and with red dots the quadrature point distribution is shown.

150 vertices were placed inside the domain, what yields 302 triangular cells. For all three approaches the same initial point set were used.

The most straight forward approach was when the points with the highest velocities ($u\_magnitde >= 0.25m/s$) got defined as vertices of the mesh. The points with more than 0.25 velocity magnitude got added to the mesh, then it was checked if the new point cloud contains any duplicate points, because duplicate vertices could yield division by zero according to code 3. Then the weights of the new vertices got set to a high number, and the initial vertices's weight got set to a small number. Either way the mesh will converge to an isotropic state, the speed of the convergence is dependent on the difference of these weights.

Vertices were placed in an optimized mesh and the initial random point cloud as well. The optimization steps were executed 100 times and the results can be seen in figure 4.9. The best density field is produced at the 10th iteration from the initially optimized mesh, the information gotten from the PINN is still visibly present, and the cell qualities are decent.

The same points were selected as before, but now they were added to the quadrature points defining the corners of the domain, since their weights are the biggest. The same

initial states were used as before, and the refinement was run for 100 cycle, the resulting meshes presented on figure 4.10. The extra quadrature points had no effect on the mesh. The refined initial mesh didn't change at all, and from it's random counter part the same mesh was produced after 100 ODT iteration.

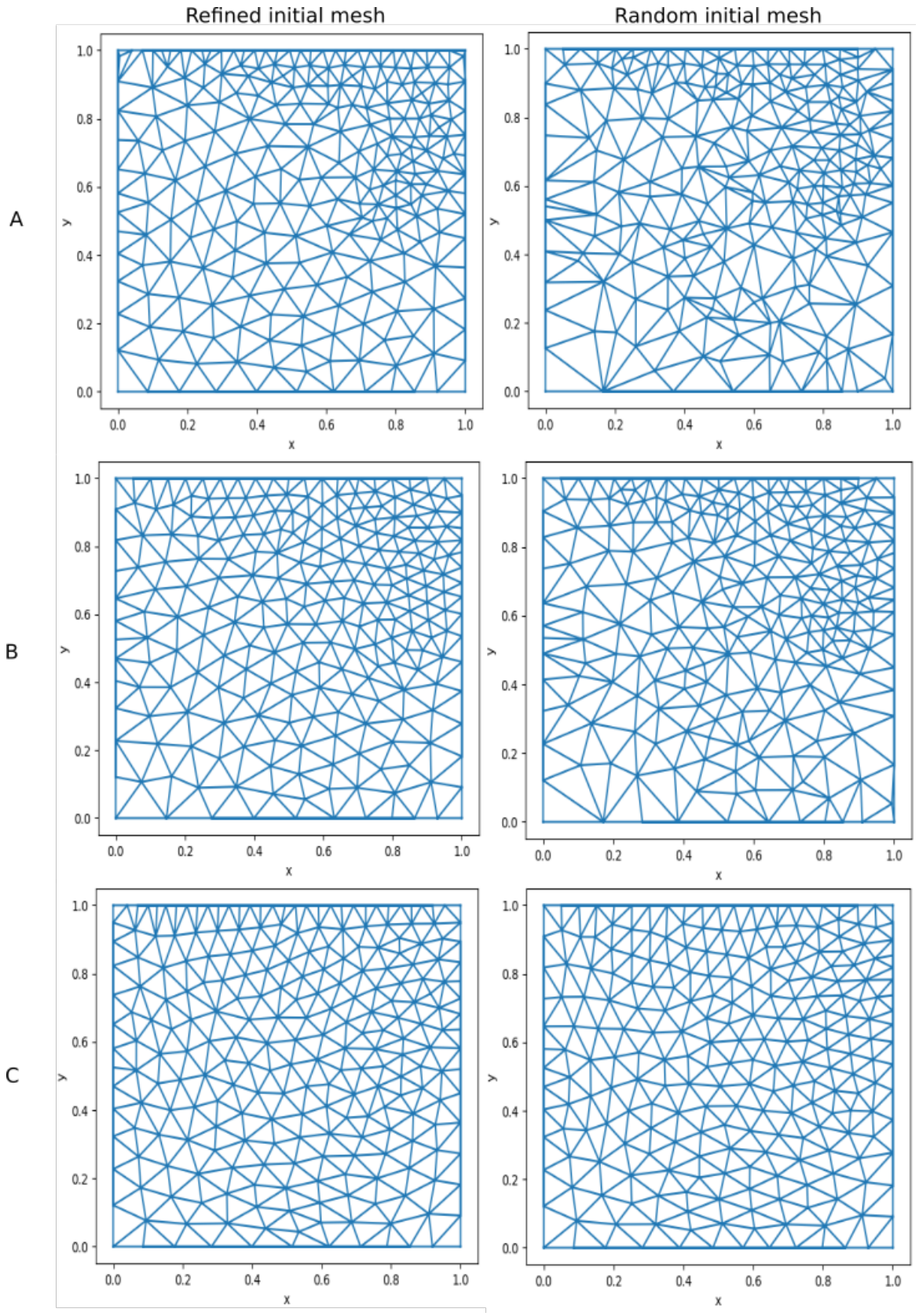

Figure 4.9: Meshes at different iterations. The grids in row A are refined with 1 iteration of ODT, row B contains the meshes after 10 refinements and row C shows the meshes after the 100th refinement. In column named 'Refined initial mesh' the points produced by PINN got placed in an initially optimized mesh, and the column 'Random initial mesh' the initial mesh haven't got refined.

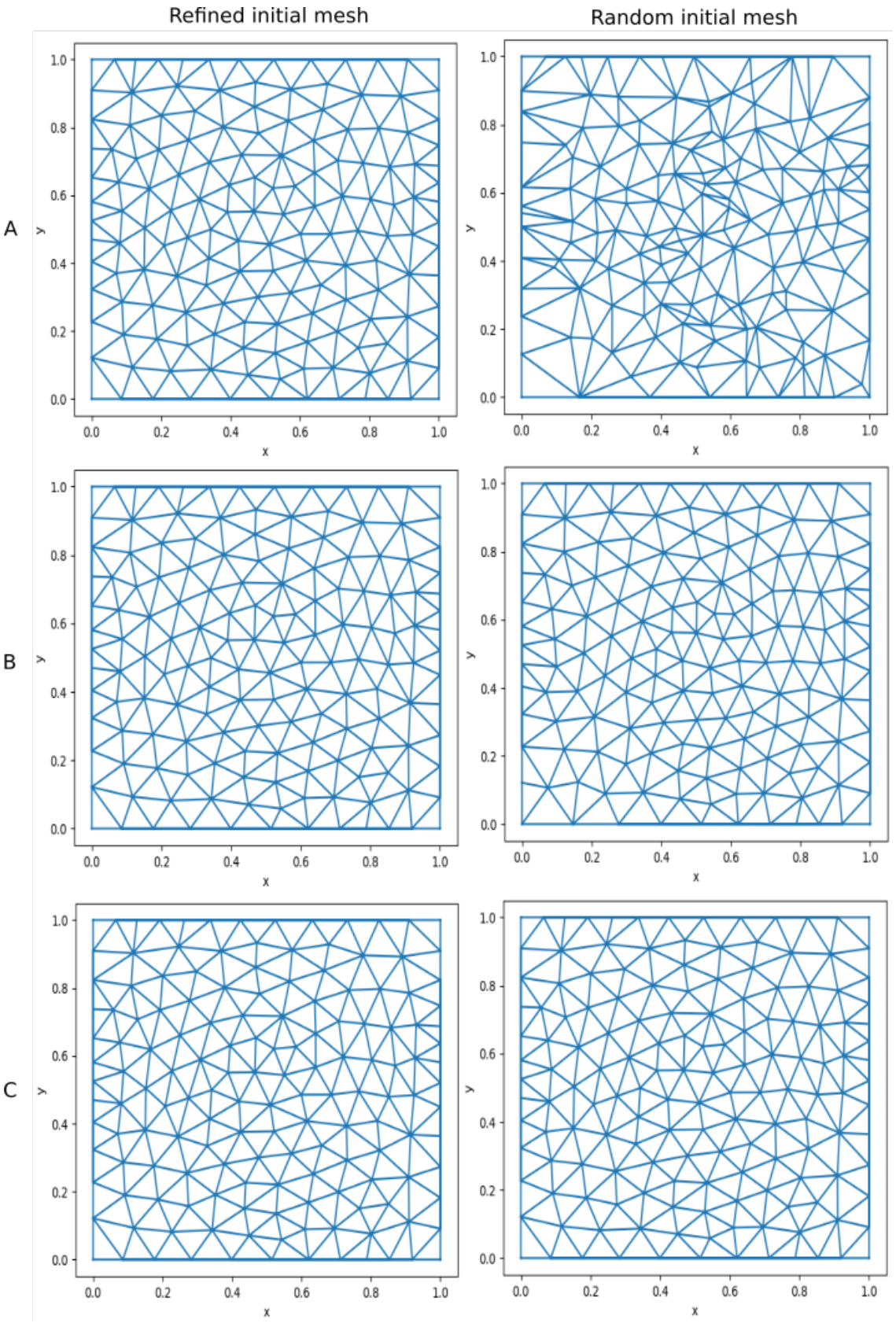

Figure 4.10: Points from PINN were defined as quadrature points. The grids in row A are refined with 1 iteration of ODT, row B contains the meshes after 10 refinements and row C shows the meshes after the 100th refinement. In column named 'Refined initial mesh' the initial state was the optimized mesh, in column 'Random initial mesh' the initial state was the mesh produced by the point cloud.

4.3.3 Extruding the mesh

The mesh "Refined initial mesh \B" was selected from 4.9. Then it got extruded with gmsh and exported as vtk a mesh, which is shown in fi. 4.11. Then with OpenFOAM's vtkUnstructuredToFoam utility the mesh was converted to poly mesh. The checkMesh utility was used to evaluate the mesh quality of the polyMesh. It has failed five criterion:

- High aspect ratio cells
- Non orthogonal faces
- Skew faces
- Zero area faces
- Zero volume cells

Most of these problems are produced by the extrusion, so with a smaller extrusion distance the same procedure was repeated, but still failed at all five criteria. Another problem was in python no surfaces were defined, and polyMesh misses these patches, which makes paraFOAM crash.

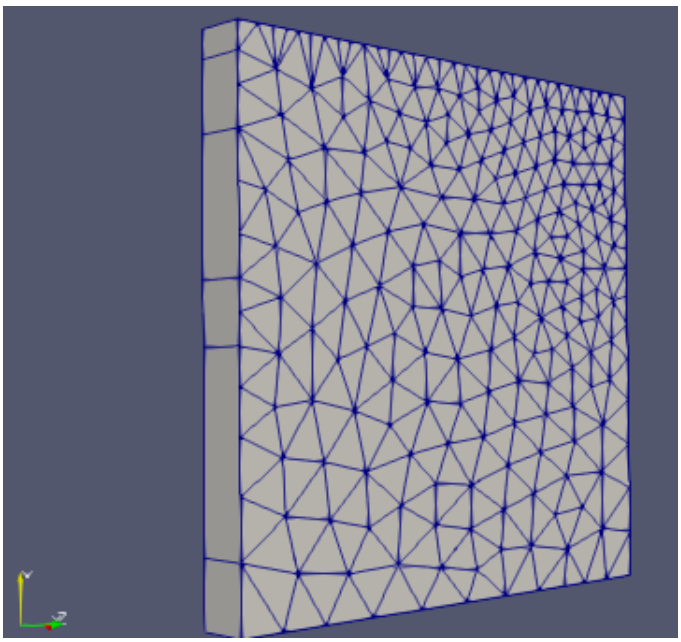

Figure 4.11: Extruded mesh visualized with paraview

4.4 Results with machine learning

4.4.1 CTGAN

CTGAN contains two neural network, which means it has a lot of parameters, and it is a derivate of GANS which requires a very big amount of data even with machine

learning standards [Lei Xu 2019]. That's why 30000 mesh were generated with the ODT implementation. All of the domains were circles but with different radius. The idea was that the network will learn how does a good mesh look like. Then from a random point set it will be able to generate an isotropic mesh.

Unfortunately the model used all of the memory available in the first epoch in google's colaboratory. Setting the vertices type to float16 didn't help, then the size of the data was reduced to 1000 meshes as well, but still had memory issues. Another potential issue could be with this idea is that the vertex positions and the domain doesn't have enough information even if the network could initialize.

4.4.2 Reinforced learning

Reinforced learning was also considered, as the promising feature of this method was that it did not require a training data set. Instead a training environment is used that contains the actions that the agent can perform, the reward function that evaluates the agent's actions, and the state space that contains states of the environment. The idea was that the functions of the meshing program would be transformed but I ran into several difficulties.

The defined problem was either too obvious or too complex. For example one approach could have been that the agent has to split a triangle cell into three other triangle cells with a placement of a vertex. The reward function would have been the sum of the aspect ratios of the new triangles, but it could be seen that the centroid of the initial cell would result in the same result. If the problem is too complex it's very hard to make the agent learn, could take thousands of hours to converge to a minimum.

Another disadvantage was that since deep reinforcement learning is relatively new field in machine learning, there is no general purpose algorithm, what can perform well in a wide variety of circumstances.

Chapter 5

Conclusion

Optimal Delaunay triangulation can indeed produce high-quality meshes in 2D, and it is relatively easy to control the sizing field of the mesh with it. But it should be noted that if a wrong function is used, then the jump between elements may increase, making the mesh unusable for finite volume methods. In this work, a physics informed neural network was used to generate a sizing field in the function of the fluid flow.

The point-velocity pairs were extracted from the flow field predicted by the neural network, then they were transformed. The velocities were scaled up and the points were transformed into mesh vertices. The meshing program was then executed on the new point cloud. The number of iterations has to be watched, if too few iterations were executed, then the mesh quality remains poor, on the other hand if the vertex positions were updated too many times, then the sizing field "disappears".

The extruded mesh quality was checked with OpenFOAM's checkMesh utility. The mesh failed the quality check. Partly it was caused by the extrusion, as there can be very small triangles and relatively large triangles as well, so it is difficult to find a good extrusion value, if it too small then the big triangles will have aspect ratio issues, if it was too big then the smaller triangles will have quality issues after the extrusion. The lack of iterations can also produce skewness and orthogonality issues in 3D.

The consideration of the initial mesh's element size and element number might prevent most of these problems, but the best solution would be to extend the method into 3D. For this it would be more appropriate to use a lower level language like C++, since even in 2D the program can run for minutes with many nodes.

Bibliography

[1] *Technical and Economic Readiness Review of CFD-Based Numerical Wave Basin for Offshore Floater Design*, volume Day 1 Mon, May 02, 2016 of *OTC Offshore Technology Conference*, 05 2016. D011S014R002.

[2] David Lee Davidson. The role of computational fluid dynamics in process industries. In *Eighth Annu Symp Front Eng*, pages 21–28, 2003.

[3] HS Pordal, CJ Matice, and TJ Fry. Computational fluid dynamics in the pharmaceutical industry. *Pharm. Technol*, 26(2):72–79, 2002.

[4] Atul Karanjkar. Design, development and optimisation using computational fluid dynamics. *Medical device technology*, 13(9):26–29, 2002.

[5] Debashis Ghosh, Patrick D Maguire, and Douglas X Zhu. Design and cfd simulation of a battery module for a hybrid electric vehicle battery pack. Technical report, SAE Technical Paper, 2009.

[6] Ippei Takaishi, Satoshi Kanai, Hiroaki Date, and Hideyoshi Takashima. Free-form feature classification for finite element meshing based on shape descriptors and machine learning. *Proceedings of CAD'19*, pages 24–26, 2019.

[7] Adel Abbas-Bayoumi and Klaus Becker. An industrial view on numerical simulation for aircraft aerodynamic design. *Journal of Mathematics in Industry*, 1(1):1–14, 2011.

[8] M. en C. Claudia García Blanquel. Algoritmo adaptativo para la selección de puntos de interés sobre estructuras bidimensionales y tridimensionales. Master's thesis, INSTITUTO POLITÉCNICO NACIONAL, 2017.

[9] Luca Lamberthi. Data driven modelling of turbulent flows using artificial neural networks. Master's thesis, Politecnico di Torino, 2019.

[10] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[11] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[12] Machine learning vs. traditional programming paradigm, 2021. [Online].

[13] Notes from the ai frontier: Applications and value of deep learning, 2021. [Online].

[14] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An introduction to computational fluid dynamics: the finite volume method*. Pearson education, 2007.

[15] what every design engineer should know about cfd, 2021. [Online].

[16] Mohd Hafiz Zawawi, A Saleha, A Salwa, NH Hassan, Nazirul Mubin Zahari, Mohd Zakwan Ramli, and Zakaria Che Muda. A review: Fundamentals of computational fluid dynamics (cfd). In *AIP Conference Proceedings*, volume 2030, page 020252. AIP Publishing LLC, 2018.

[17] Chenguang Song, Yuan Zheng, Zhenzhou Zhao, Yuquan Zhang, Chong Li, and Hao Jiang. Investigation of meshing strategies and turbulence models of computational fluid dynamics simulations of vertical axis wind turbines. *Journal of Renewable and Sustainable Energy*, 7(3):033111, 2015.

[18] Tomás Norton and Da-Wen Sun. Computational fluid dynamics (cfd)–an effective and efficient design and analysis tool for the food industry: a review. *Trends in Food Science & Technology*, 17(11):600–620, 2006.

[19] Jiannan Tan. *A study of solving Navier-Stokes equations with a finite volume method based on polygonal unstructured grids and the computational analysis of ground vehicle aerodynamics*. PhD thesis, Texas Tech University, 2010.

[20] Introduction of computational fluid dynamics, 2021. [Online].

[21] Zheng Zheng Hu, DM Causon, CG Mingham, and L Qian. Numerical simulation of floating bodies in extreme free surface waves. *Natural Hazards and Earth System Sciences*, 11(2):519–527, 2011.

[22] Claudio Lobos, Yohan Payan, and Nancy Hitschfeld. Techniques for the generation of 3d finite element meshes of human organs. *Informatics in Oral Medicine: Advanced Techniques in Clinical and Diagnostic Technologies*, pages 126–158, 2010.

[23] Reporting mesh statistics, 2021. [Online].

[24] Can a wrong mesh impact your simulation results?, 2021. [Online].

[25] 3d számítógépes geometria és alakzatrekonstrukció háromszöghálók, 2021. [Online].

[26] Voronoi diagram, 2021. [Online].

[27] Kevin Wilson, Gerard Guiraudon, Douglas L Jones, and Terry M Peters. Mapping of cardiac electrophysiology onto a dynamic patient-specific heart model. *IEEE transactions on medical imaging*, 28(12):1870–1880, 2009.

[28] Qiang Du and Desheng Wang. Tetrahedral mesh generation and optimization based on centroidal voronoi tessellations. *International journal for numerical methods in engineering*, 56(9):1355–1373, 2003.

[29] Pierre Alliez, David Cohen-Steiner, Mariette Yvinec, and Mathieu Desbrun. Variational tetrahedral meshing. In *ACM SIGGRAPH 2005 Papers*, pages 617–625. 2005.

[30] David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. Variational shape approximation. In *ACM SIGGRAPH 2004 Papers*, pages 905–914. 2004.

[31] Long Chen. Mesh smoothing schemes based on optimal delaunay triangulations. In *IMR*, pages 109–120. Citeseer, 2004.

[32] Variational tetrahedral meshing, 2021. [Online].

[33] M. S. Smit and W. Bronsvoort. Variational tetrahedral meshing of mechanical models for finite element analysis. *Computer-aided Design and Applications*, 5:228–240, 2008.

[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[35] Supervised and unsupervised machine learning algorithms, 2021. [Online].

[36] Simon Bence Szilárd. Deep learning architektÚrÁk alkalmazÁsa energetikai idŐsorok elŐrejelzÉsÉben. Master's thesis, Budapesti Műszaki és Gazdaságtudományi Egyetem, 2017.

[37] 2021. [Online].

[38] 2021. [Online].

[39] Császár Noel. Deep learning optimalizÁciÓs mÓdszerek vizsgÁlata. Master's thesis, Budapesti Műszaki és Gazdaságtudományi Egyetem, 2019.

[40] 2021. [Online].

[41] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. Visualizing higher-layer features of a deep network. *University of Montreal*, 1341(3):1, 2009.

[42] Borza Marcell. Mesterséges neurális hálózatok matematikai alapjai. Master's thesis, Eötvös Loránd Tudományegyetem, 2019.

[43] 2021. [Online].

[44] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[45] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[46] Larry Manevitz, Malik Yousef, and Dan Givoli. Finite–element mesh generation using self–organizing neural networks. *Computer-Aided Civil and Infrastructure Engineering*, 12(4):233–250, 1997.

[47] Olivier Sarzeaud, Yann Stephan, and Claude Touzet. Finite element meshing using kohonen's self-organizing maps. In *Artificial Neural Networks*, pages 1313–1317. Elsevier, 1991.

[48] DA Lowther and W Mai. On automatic mesh generation using kohonen maps. *IEEE transactions on magnetics*, 34(5):3391–3394, 1998.

[49] S Alfonzetti, S Coco, S Cavalieri, and M Malgeri. Automatic mesh generation by the let-it-grow neural network. *IEEE transactions on magnetics*, 32(3):1349–1352, 1996.

[50] Çinar Ahmet and Arslan Ahmet. Neural networks based mesh generation method in 2-d. In *Eurasian Conference on Information and Communication Technology*, pages 395–401. Springer, 2002.

[51] S Yao, B Yan, B Chen, and Yong Zeng. An ann-based element extraction method for automatic mesh generation. *Expert Systems with Applications*, 29(1):193–206, 2005.

[52] Zheyan Zhang, Yongxing Wang, Peter K Jimack, and He Wang. Meshingnet: A new mesh generation method based on deep learning. In *International Conference on Computational Science*, pages 186–198. Springer, 2020.

[53] 2021. [Online].

[54] 2021. [Online].