EPFL

Thèse n° 8439

Tessellation and Improvement of Simplicial Meshes using Neural Networks

Présentée le 3 juin 2021

Faculté des sciences et techniques de l'ingénieur
Laboratoire de mécanique des fluides et instabilités
Programme doctoral en mécanique

pour l'obtention du grade de Docteur ès Sciences

par

Alexis PAPAGIANNOPOULOS

Acceptée sur proposition du jury

Prof. W. Curtin, président du jury
Prof. F. Gallaire, Dr P. Clausen, directeurs de thèse
Prof. D. Triantafyllidis, rapporteur
Dr L. Ladicky, rapporteur
Prof. L. Villard, rapporteur

École
polytechnique
fédérale
de Lausanne

2021

To my parents, brother and wife . . .

Acknowledgements

This is the occasion to thank great leaders, guides, colleagues, and friends:

Prof. Avellan, who always kept a sound scientific critical approach in my work and made this possible with his support and wisdom.

Dr. Clausen, for constantly leading me to put my fullest potential at use and for closely following up with all the details of our work.

Prof. Curtin, for being supportive and open to help all the time, as well as accepting to be a member of the Juries.

Prof. Gallaire, for his welcoming and great collaborative spirit. Even if our partnership was short, I will always appreciate his quick and dynamic actions.

Prof. Villard, Prof. Triantafyllidis and Dr. Ladicky for taking interest in my doctoral thesis work and agreeing on evaluating it.

Mr. Flynn, for his outstanding talent and his unbreakable will to continue working, regardless of the adverse period we all faced.

The whole “old” LMH group and the LFMI group, specially to Isabelle and Petra Erika – Your guidance, advice and understanding was crucial, thank you so much!

Lastly, this work is dedicated to my parents, my brother, my wife, the memory of my grandparents and my friends. Each one for them kept me going with their constant support and good attitude.

Lausanne, February 8, 2021 A. P.

Abstract

Modern mesh generation addresses the development of robust algorithms that construct a discrete representation of the geometry into polytopal elements conforming to divergent properties: (i) fidelity to complex geometrical features, (ii) support for high spatial resolution in areas of interest and sparsity elsewhere, and (iii) preservation of optimal element geometry (quality).The automation of the meshing process with respect to these properties is still considered a critical bottleneck as it is often tied to the development of complex algorithms; although such algorithms produce meshes that satisfy desirable properties, they may entail a significant computational cost. To tackle the automation hurdles of current algorithms, this research work studies the adaption of Neural Networks (NNs) that have been proven efficient in automating complex problems, for the development of meshing algorithms.

A machine learning meshing scheme for the generation of simplicial meshes is proposed based on the predictions of NNs. The scheme is applied to small contours with up to 16 edges. The data extracted from the meshed contours are utilized to train NNs that approximate the number of vertices to be inserted inside a contour cavity, their location, and the connectivity. Based on an element quality metric, the results show a maximum deviation of 27.3% on the minimum quality between the elements of the meshes generated by the scheme and the ones generated from a reference mesher. This level of deviation corresponds to produced meshes with element angles that lie between $28° \leq \theta \leq 106°$ in the worst case. Such results validate the use of the developed scheme for good quality mesh generation.

The trained NNs of the meshing scheme, along with a set of NNs that reposition vertices of a mesh, are used to develop a machine learning based mesh improvement algorithm that applies operations to improve the quality of elements. The efficiency of the operations is validated and evaluated after their inclusion to local mesh improvement schemes that are applied to: (i) perturbed static meshes containing low quality elements and (ii) dynamic meshes that are subjected to simulations. The operations improve the quality for all test cases at a reduced computational cost when compared to existing operations. In the worst case, the application of the improvement schemes result in static meshes with element angles between $31° \leq \theta \leq 109°$ and in dynamic meshes with minimum and maximum angles that lie between $35° \leq \theta_{min} \leq 44°$ and $90° \leq \theta_{max} \leq 108°$, respectively, during the course of the simulation.

Finally, an iterative machine learning based scheme is developed to mesh larger uniform and adaptive element size meshes. Based on a high-resolution contour that represents the boundaries of a geometry, an initial mesh for a low-resolution contour is created using the meshing scheme developed for small contours. Next, vertices are inserted in the edges of the elements and projected to the high-resolution contour. Each sub-contour created after this process is meshed using the meshing scheme. The quality of the elements is improved using the developed mesh improvement algorithm. The application iterates until an element target edge length is achieved. Examples of meshed geometries represented by high resolution contours with up to 201 edges, demonstrate that the application produces good quality uniform and adaptive meshes containing up to $1,658$ elements with angles laying between $28^\circ \leq \theta \leq 111^\circ$.

Keywords: Simplicial Mesh, Mesh Generation, Mesh Improvement, Neural Networks, Machine Learning

Résumé

La génération de maillages concerne le développement d'algorithmes robustes qui construisent une représentation discrète de la géométrie en éléments polytopaux conformant à des propriétés divergentes : (i) fidélité aux caractéristiques géométriques complexes, (ii) support d'une haute résolution spatiale dans des zones d'intérêt et d'une faible densité ailleurs, et (iii) préservation de la géométrie optimale des éléments (qualité). L'automatisation du processus de génération de maillage respectant ces propriétés est souvent considérée comme un goulot d'étranglement car elle est souvent liée au développement d'algorithmes complexes ; bien que ces algorithmes produisent des maillages satisfaisant des propriétés désirables, ils peuvent entraîner un coût computationnel important. Afin de surmonter les obstacles à l'automatisation des algorithmes actuels, ce travail de recherche étudie l'adaptation de réseaux de neurones artificielles (RNs) qui ont été prouvés efficaces pour automatiser des problèmes complexes, pour le développement d'algorithmes de maillage.

Un schéma d'apprentissage automatique pour la génération de maillage simpliciaux est proposé selon les prédictions de RNs. Le schéma est appliqué aux petits contours qui contiennent jusqu'à 16 arêtes. Les données extraites des contours maillées sont utilisées pour entraîner les RNs qui prédisent le nombre de sommets à insérer dans la cavité des contours, leurs positions, et la connectivité. Basée sur une métrique de qualité d'éléments, les résultats démontrent une déviation maximale de 27.3 % sur la qualité minimale entre les éléments des maillages générés par le schéma développé et ceux des maillages générés par le schéma de référence. Ce niveau de déviation correspond à des maillages ayant des angles d'éléments compris entre $28^\circ \leq \theta \leq 106^\circ$, au pire des cas. De tels résultats valident l'utilisation du schéma pour générer des maillages de bonne qualité.

Les RNs entraînés pour le schéma de maillage, ainsi qu'un ensemble de RNs repositionnant les sommets du maillage, sont utilisés pour développer un algorithm d'apprentissage automatique d'amélioration de maillage, qui applique des opérations pour améliorer la qualité des éléments. L'efficacité des opérations est évaluée et validée à la suite de leur inclusion dans les schémas locaux d'amélioration de maillage qui sont appliqués à : i) des maillages statiques perturbés contenant des éléments de faible qualité, et ii) des maillages dynamiques qui sont sujet à des simulations. Ces opérations améliorent la qualité pour tous les cas tests à un coût computationnel réduit comparativement aux opérations existantes. Au pire des cas, au cours de la simulation, l'application des schémas d'amélioration dérivent des maillages statiques ayant des angles compris entre $31^\circ \leq \theta \leq 109^\circ$ et des maillages dynamiques avec des angles minimum and maximum compris entre $35^\circ \leq \theta_{min} \leq 44^\circ$ et $90^\circ \leq \theta_{max} \leq 108^\circ$,

respectivement.
Finalement, un schéma itératif d'apprentissage automatique est développé afin de mailler de plus larges maillages de taille d'éléments uniforme et adaptative. Basé sur un contour de haute résolution représentant les extrémités d'une géométrie, un maillage initial pour un contour de basse résolution est créé en utilisant le schéma de maillage développé pour des petits contours. Ensuite, des sommets sont insérés aux arêtes des éléments et projetés aux contours de haute résolution. Chaque sous-contour, créé après ce processus, est maillé en utilisant le schéma de maillage. La qualité des éléments est améliorée avec l'algorithme développé d'amélioration de maillage. L'application itère jusqu'à ce qu'une taille d'arête cible est atteinte. Des exemples de géométries de maillage représentés par des contours de haute résolution jusqu'à 201 arêtes, démontrent que l'application produit des maillages uniformes et adaptifs de bonne qualité contenant jusqu'à 1,658 éléments et des angles compris entre $28° \leq \theta \leq 111°$.

Keywords : Maillage Simplicial, Génération de Maillage, Amélioration du Maillage, Réseaux de Neurones, Apprentissage Automatique

Contents

Acknowledgements **i**

Abstract (English/Français/Deutsch) **iii**

1 Introduction **1**

Introduction **1**

1.1 Motivation 1
1.2 Meshes 2
1.3 Mesh Generation algorithms 6
  1.3.1 Quadtree/Octree 6
  1.3.2 Advancing front 8
  1.3.3 Delaunay methods 9
  1.3.4 Hybrid methods 14
1.4 Mesh improvement 15
1.5 Artificial Neural Networks 21
  1.5.1 Training process 25
  1.5.2 Hyperparameters 28
  1.5.3 Convolution 32
1.6 State of the art 34
  1.6.1 Automatic mesh generation 34
  1.6.2 Mesh Improvement algorithms 35
  1.6.3 Machine learning and meshes 38
1.7 Research Objective 43
1.8 Outline 44

2 Meshing of 2-D simplicial contours using Neural Networks **47**

2.1 Problem Statement 47
2.2 Algorithm overview 48
2.3 Feature transformation and training data acquisition 50
2.4 Prediction of the number of inner vertices 53
2.5 Prediction of the inner vertices positions 54
2.6 Prediction of the connectivity 56
  2.6.1 Triangulation algorithm 56

2.6.2 Grid sampling augmentation of the inner vertices 62
2.6.2.1 Adaptive sampling strategy 62

3 Results and validation of 2-D simplicial contour Meshing using Neural Networks 65
3.1 Experimental Conditions 65
3.1.1 Error metrics 65
3.1.2 Training dataset populations 65
3.1.3 NN_i hyperparameters 66
3.2 Results 68
3.2.1 Predictions of the number of inner vertices 68
3.2.2 Prediction of the inner vertices positions 70
3.2.3 Prediction of the connectivity 73
3.2.4 Efficiency of adaptive sampling 76
3.2.5 Overall prediction of the meshing scheme 78
3.3 Conclusions 78

4 2-D Local Mesh Improvement using Neural Networks 83
4.1 Local Mesh Improvement Operations 84
4.1.1 Reconnection 84
4.1.2 Vertex Repositioning 85
4.1.3 Surface control 87
4.1.4 Size control 88

5 Results and validation of 2-D Local Mesh Improvement using Neural Networks 91
5.1 Experimental parameters and results 91
5.1.1 NN hyperparameters and training populations 91
5.1.2 Experiments 92
5.1.2.1 Static Mesh Improvement 94
5.1.2.2 Dynamic Mesh Improvement 100
5.2 Conclusions 110

6 Meshing large meshes 113
6.1 Scheme for large mesh generation with uniform element size 113
6.2 Scheme for large mesh generation with adaptive element size 115

7 Conclusion and outlook 119
7.1 Conclusion 119
7.2 Outlook 122

Bibliography 125

Bibliography 134

A Supplementary material 135

A.1 Back propagation 135
A.2 *Gmsh*© mesh generation 137
A.2.1 Contour mesh generation 137
A.2.2 Test cases mesh generation 138
A.3 Triangulation algorithm 142
A.3.1 Locking mechanism 142
A.3.2 Sub-contour detection 142

B Code 145
B.1 Mesh generation 145
B.1.1 Feature Transformation 145
B.1.2 Approximation of inner vertices location 146
B.1.2.1 Point selection 146
B.1.2.2 Interpolation 149
B.1.3 Triangulation algorithm 150
B.1.3.1 Vertex locking mechanism 150
B.1.3.2 Spotting sub-contours 152
B.1.3.3 Triangulation 165
B.2 Mesh Improvement 173
B.2.1 Mesh class 173
B.2.2 Reconnection 176
B.2.3 Vertex repositioning 179
B.2.4 Boundary/interface vertex repositioning 181
B.2.5 Edge Length control 185
B.3 Large Mesh generation 191

Curriculum Vitae 201

List of Figures

1.1 A 2D triangular cap element with a large angle (a) and a 3D sliver tetrahedral element (b) with large dihedral angles. These type of elements introduce large errors to the gradients of the basis functions. 3

1.2 Examples of a structured mesh (a) and an unstructured mesh (b). 5

1.3 (a) The original geometry (b) A bounding box occludes the geometry and cells are inserted. Smaller cells are inserted for a better approximation of more complex regions. (c) The cells are meshed according to templates. (d) The final mesh after snapping the vertices of boundary cells to vertices of the geometry's boundary and cutting elements with vertices outside the geometry domain. 6

1.4 (a) The boundary of the geometry is discretized according to a user defined edge length. An edge (facet) is connected to a vertex P_{opt} to form an element. A circle of radius r centered at P_{opt} is used to spot possible intersections with other fronts or to check if other vertices are included in it. (b) A vertex of another front is spotted inside the circle of the candidate vertex. (c) In this case the original candidate vertex is rejected and the vertex that belongs to the neighbor front is selected to form a new element. (d) The formation of the front after several stages of the method. (e) The final mesh. 8

1.5 Example of Delaunay criterion. (a) maintains the criterion while (b) does not. . 9

1.6 From left to right: A vertex P is inserted in a triangulation The vertex is included into the circumcircle of the shaded triangular elements The elements whose circumcircle include P are deleted and the vertices of the countour cavity formed after the deletion are connected with P. 10

1.7 Steps of the CDT algorithm. (a) A bounding box composed of two simplices (triangles) occludes the vertices of the discretized boundary of the geometry. (b) An initial triangulation is performed using the Boywer-Watson algorithm. The initial triangulation contains the vertices of the discretized boundary and the vertices of the boundary box. (c) After the initial triangulation, the boundary facets are recovered and the elements that are located outside of the geometry are deleted (d) Delaunay refinement inserts new vertices to strategic locations of the initial triangulation to improve the size and shape of the elements. 10

1.8 (a) A constrained edge (highlighted) is encroached if a vertex is contained within its diametral circle (Ruppert (1993)). The constrained edge is split until no vertices are included in the diametral circle no constrained edge is encroached. (b) Alternatively an encroached constrained edge can be spotted if a badly shaped element (t) and its circumcenter (c) lie on opposite sides of it (Chew (1989)). All vertices in the encroached constrained edge's diametral circle are deleted and a point is inserted in the middle of it to form new elements. 13

1.9 Example of smoothing operation. The vertex is repositioned to improve the quality of elements that are connected to it. 15

1.10 Examples of flip operator in 2D and 3D. In 2D, the flip operator changes the connectivity by swapping the adjacent edge of two elements (a). In 3D, there are several variants of the flip operator: the 2-3, 3-2 flip operators (b) and the 2-2, 4-4 flip operator (c) where the numbers denote the number of tetrahedra before and after applying the operation. 16

1.11 Edge removal and multi-face removal operations. Edge removal triangulates the domain R that contains the ring vertices around the edge ab which is an common edge for the tetrahedra in I. After the triangulation T the faces are connected with the vertices a and b to form the new set of tetrahedra J. Multi-face removal adjoins the sandwiched faces of the set of tetrahedra J, connects the a and b to form an edge, and connects the ring of vertices R with a and b to form the new set of tetrahedra I. 16

1.12 Example of vertex cavitation. The operation views the mesh as graph with nodes that correspond to the elements and directed edges that correspond to facet adjacent elements with a parent-child relation. Starting from the elements of p and following the directed edges, adjacent elements are visited to check whether they are included in the formation of a cavity C that includes p. The vertices of C are connected to p to form new elements. 17

1.13 Vertex insertion operation. (a) Example of edge insertion. An vertex is inserted in the middle of an edge. The vertices of the element are connected to the inserted vertex to form new elements. (b) Example of tetrahedral insertion. A vertex is inserted in the barycenter of the tetrahedron and the vertices of the tetrahedron are connected with it. 18

1.14 Example of edge contraction. An element is removed by collapsing its short edge. 18

1.15 Neural network with feed forward architecture. Neurons are grouped into a sequence of c layers $L^{[1]}, ..., L^{[c]}$. The hidden layers $L^{[1]}, ..., L^{[c-1]}$ contain $h_1, ..., h_{c-1}$ hidden neurons, and the output layer $L^{[c]}$ is composed of m output neurons. Each layer is associated with a set of free parameters $a = (a^{[1]}, .., a^{[c]})$. For every layer l the free parameters are defined as the pair $a^{[l]} = (w^{[l]}, b^{[l]})$, where $w^{[l]} = (w^{[l]}_{1,1}, w^{[l]}_{1,2}, .., w^{[l]}_{1,h_l}, ..., w^{[l]}_{h_{l-1},1}, ..., w^{[l]}_{h_{l-1},h_l})$ are the weight synapses, $b^{[l]} = (b^{[l]}_1, .., b^{[l]}_{h_l})$ are the biases, and h_l is the number of neurons of the $L^{[l]}$ layer. The output of each neuron is an output of a non linear function $y^{[l]}_j$, where $l \in \{1, 2, ..., c\}$, $j \in \{1, 2, ..., h_l\}$. 22

1.16 Perceptron neuron model. The input signals $X^{[l-1]} = (x_1^{[l-1]}, .., x_{h_{l-1}}^{[l-1]})$ from $L^{[l-1]}$ layer are transformed using the activation function to the signal $u_j^{[l]} = b_j^{[l]} + \sum_{i=1}^{h_l} w_{i,j}^{[l]} x_i^{[l-1]}$. The outcome of $u_j^{[l]}$ is then given as input to the activation function $g^{[l]}$ that defines the output signal $y_j^{[l]}$. The output signal $y_j^{[l]}$ becomes an input signal $x_j^{[l]}$ for neurons of the $L^{[l+1]}$ layer. 23

1.17 Graphs of the logistic, hyperbolic, and ReLU activation functions. 24

1.18 Example of gradient descent for fitting a giving dataset $(X^{(i)}, Y^{(i)})$, where $i = \{1, 2, ..m\}$, to a straight line $h_a(x) = w_0 + w_1 x$. The free parameters $a = (w_0, w_1)$ are updated through the iterative process $w_{j,t+1} = w_{j,t} - \eta(\partial \mathscr{L}(w_t)/\partial w_j)$, where $j = \{0, 1\}$, using the objective functional $\mathscr{L}(w_t) = (1/2m)\sum_{i=0}^{m}(h_a(X^{(i)} - Y^{(i)})^2$. At each iteration the free parameters a get closer to the minima of the objective functional and the line fits better the dataset. 26

1.19 Optimization path using batch gradient descent, SGD and mini batch SGD for a convex loss function. Batch gradient descent, although computationaly expensive, is proven to converge to the minimal of convex loss funtions. SGD is faster, however, due to its nature, the optimization path may oscillate and overshoot near the minima. Mini batch SGD can be computationally efficient and can reduce the oscillations of SGD. 29

1.20 Optimization path using mini batch SGD with different optimization strategies. Momentum and Adam add an additional computational cost at each free parameter update step but reduce the oscillating behavior of mini batch SGD making it possible to reach faster the minima. 31

1.21 (a) Example of a convolutional NN. Multiple kernels are used to apply convolution to the input signal resulting in multiple convoluted feature maps. Each of convoluted feature map undergoes a pooling process. The process of convolution and pooling can repeat before flattening the result and connecting it with an NN. (b) Example of convolution and pooling operation to an input signal of signal $n \times 2$. Using a kernel K of size 2×2 with a stride $\mathscr{F}_K = 1$ the convoluted signal has a size $(n-1) \times 2$. Applying the max pooling layer with a stride $\mathscr{F}_p = 1$ results in a signal $(n-2) \times 2$. 32

2.1 (a) A set of points and edges defines a closed boundary ∂V (contour) with an interior continuous domain V (cavity). (b) To form a good quality mesh consisting of simplicial (triangular) elements whose edges respect a specific length, vertices are inserted in strategic locations of the interior domain V and are connected resulting in the discretization of V into $V_i, i = 1, 2, ...N_{el}$ triangular elements, where N_{el} is the number of elements such that $\cup_{i=1}^{N_{el}} V_i = V$. The intersection of sub domains $V_i \cap V_j$ is at most an element edge; the tessellated domain does not contain intersections between the elements of the mesh. 47

2.2 The meshing scheme consists of four steps: (i) The initial contour with N_C edges is scaled and rotated with respect to a regular polygon with N_C edges inscribed in a unitary radius circle. (ii) NN_1 is used to approximate the number of inner points N_I, providing as input the contour vertex coordinates P_C and the requested target edge length l_s. (iii) NN_2 takes as input the vertex coordinates P_C, patches of grid points from G, and the target edge length l_s. It outputs the scores S_G for each grid point. Based on S_G, N_I grid points are selected and interpolation is applied to a region around them. Next, to approximate the inner vertices P_I, the local minimum of the interpolated surface are found. (iv) NN_3 takes as input the contour vertices P_C and the inner point vertices P_I and outputs the entries of a connection table A. The contour is meshed with a triangulation algorithm that meshes the cavity of the contour based on A. After the termination of the algorithm, if a sub-contour with $P_C^{'}$ contour coordinates is created containing $N_I^{'}$ inner vertices with $P_I^{'}$ coordinates, NN_3 is called recursively to mesh the sub-contour, until no further sub-contour emerges. 49

2.3 From left to right: Steps followed to acquire the mesh of a contour. Step (i) of the algorithm consists of a feature transformation applied to the contour that causes the scaling of the target elements size of edge length l to l_s. Next, following steps (ii) and (iii) of the proposed meshing scheme, based on the prediction of NN_1, one vertex is inserted in the interior of the contour (cavity), and its location is predicted using NN_2. The final step (iv) of the meshing scheme uses NN_3 and a triangulation algorithm to connect the edges of the contour with inner vertices or contour vertices to create the mesh. 50

2.4 Procrustes superimposition on contours with 6 edges with requested element size l. The reference contour is a regular hexagon inscribed in a unit circle with coordinates $Q = \{q_i, i = 1,2,..,6\}$. The contour is scaled by a scale factor S, changing the target edge length from l to $l_s = Sl$, and rotated to best fit the point of the reference polygon to acquire the points of the transformed contour $P_C = \{p_i, i = 1,2,..,6\}$. 51

2.5 Example of creation of contour with 6 edges. A unit circle is divided into 6 sectors. From each sector a point $p_i, i = \{1,..,6\}$ is selected. To avoid the creation of contours with very short edges, no points are selected from the inner region of a circle with small radius r. 52

2.6 The generated contours are meshed by applying CDT followed by refinement for various target edge lengths l_s producing a graded mesh. The number of inner vertices N_I and their coordinates P_I from the graded mesh are used to train the NN_1 and NN_2. By knowing the location P_I of the inner vertices, the connection table A of the contour is calculated to be included in the training dataset of NN_3. 52

2.7 NN_1 architecture for the prediction of number of vertices. It takes as input the contour coordinates $P_C = \{p_i = (x_i, y_i), i = 1,2,..,N_C\}$ and the scaled target edge length l_s. NN_1 outputs the approximation $\hat{N}_I$ of the number of inner vertices that should be inserted inside the cavity of the contour to achieve the target edge length l_s. 53

2.8 Example of the NN_2 approximation of two inner vertices $p_{I,1}$ and $p_{I,2}$ ($N_I = 2$) for a contour with 8 edges. (a) Based on the scores S_G, the grid points $\hat{p}_{G_1}$ and $\hat{p}_{G_2}$ are selected as the first two grid points with the minimum score. (b) Then, interpolation is applied to a local region around them. Here, interpolation is applied to find the scores on a region around $\hat{p}_{G_1}$. This region includes the grid points around $\hat{p}_{G_1}$. The number of grid points included in the region depends on the target edge length l_s. By locating the local minimum of the interpolated surface, the approximation $\hat{p}_{I,1}$ of $p_{I,1}$ is acquired. (c) The interpolation procedure is also applied to $\hat{p}_{G_2}$ to obtain $\hat{p}_{I,2}$. 54

2.9 The grid G defined over the contour is divided into N_p patches G_k, $k = \{1,2..N_p\}$ (here $N_p = 4$). NN_2 takes as input the contour coordinates $P_C = \{p_i, i = 1,2,..,N_C\}$, the coordinates of the N_{G_k} grid points $P_{G_k} = \{p_{G_{(i,j)}} : (i,j) = \{1,2,..,a\} \times \{1,2,..,a\}, a^2 = N_{G_k}\}$ that are included inside a patch, and the target edge length l_s. It outputs the scores $\hat{s}_{i,j}$ that correspond to each grid point inside the patch. 55

2.10 The cavity is meshed according to the entries of the connection table. Here, the connection table contains the values of q_{worst}. Each facet (row) is connected with the vertex (column) that has the maximum value of q_{worst}. First, each row is ordered by increasing quality. Subsequently, the columns are ordered with the same criteria. Once the connection table is sorted, the meshing algorithm is called. When computing the connection table, the connection entry of a facet with a vertex that forms a element outside the cavity of contour (e.g $\{p_2, p_3, p_4\}$) is omitted to zero by computing the signed area of the element. Valid elements have a positive signed area, while invalid have a negative signed area. In the depicted example, the triangulation algorithm starts by connecting the facet $\{p_1,p_2\}$ with the vertex p_3 to create the element $\{p_1,p_2,p_3\}$. This connection is done with accordance to the higher value of the row of the connection table (i.e 0.66). The creation of $\{p_1,p_2,p_3\}$ locks the vertex $\{p_2\}$ for any further connections. Since $\{p_2,p_3\}$ is another contour facet included in the formed element, $\{p_2,p_3\}$ is tagged as locked (F_{locked}). This removes the row $\{p_2,p_3\}$ and the algorithm proceeds to connect facet $\{p_2,p_3\}$ with p_6. In a similar fashion, all the facets of the contour are connected with the vertex that contains the highest entry to form the mesh incrementally, one element after the other. 57

2.11 (a)-(e) Example of meshing a 2-D cavity while the sets V_{locked} and F_{locked} are updated. (a) First, the facet (edge) $\{p_3,p_4\}$ is connected to vertex p_5. The creation of the element $\{p_3,p_4,p_5\}$ locks vertex p_4, as it can no longer be connected with another facet. The facet $\{p_4, p_5\}$ is also locked as it can no longer connect with another vertex. V_{locked} now contains vertex p_4 and $\{p_4, p_5\}$ is tagged as F_{locked}. (b) Facet $\{p_2,p_3\}$ connects with vertex p_6 creating the element $\{p_2,p_3,p_6\}$. (c) The creation of element $\{p_2,p_3,p_6\}$ causes the apparition of element $\{p_3,p_5,p_6\}$. V_{locked} will be updated with vertices p_3 and p_5 and facet $\{p_5, p_6\}$ will be tagged as F_{locked}. (d) By connecting the facet $\{p_1,p_2\}$ with vertex p_7 the element $\{p_1,p_2,p_7\}$ locks vertex p_1 and tags facet $\{p_1, p_7\}$ as F_{locked}. (e) The creation of element $\{p_1,p_2,p_7\}$ causes the apparition of element $\{p_2,p_6,p_7\}$ that locks vertices p_2 and p_6. All vertices are now included in V_{locked} (termination of the algorithm) . . . 58

2.12 (a) Example of the appearance of a sub-cavity with a single element. After a first iteration of the triangulation algorithm the facets of the contour are connected with the vertex that corresponds to the highest entry of the connection table. This also causes the appearance of the sub-cavity containing the element $\{p_2,p_4,p_6\}$. The element is added to the list of elements to terminate the triangulation process. (b) Example of sub-cavity that forms a contour. After a first iteration of the triangulation algorithm, once all facets of the contour are connected to the vertex that corresponds to the highest entry, the contour $\{p_2,p_4,p_6,p_8,p_{10}\}$ is formed. In this case, the triangulation algorithm is called recursively to mesh the new contour. (c) Example of sub-cavity that forms a contour that contains inner points. After a first iteration of the triangulation algorithm, the contour $\{p_0,p_2,p_4,p_6,p_8,p_{10}\}$ is formed that contains the inner points $p_{I,1}$ and $p_{I,2}$. Similarly, the triangulation algorithm is called recursively to mesh the contour with the inner points. 59

2.13 NN_3 starts by applying 2-D convolution to the coordinates P_C that are ordered in a circular way. It proceeds by applying a pool function to the convoluted result. The flattened outcome of pooling along with the coordinates of the inner vertices P_I are then connected with multilayer perceptrons. It outputs the entries $a_{i,j}$ of the connection table. 61

2.14 Example of sampling inner vertices for the training of NN_3 for $N_I = 3$. From a grid of inner vertices, $p_{I,1}$ is randomly chosen and all the vertices contained at a distance of $0.1 l_s$ from it are excluded from being selected later on. $p_{I,2}$ is then selected, imposing the same exclusion zone, and finally $p_{I,3}$. All the vertices from the grid are at a distance of $0.1 l_s$ from the edges of the contour. 62

2.15 (a) For the highlighted edge, the worst mesh quality is calculated by connecting this edge with each point of the grid. (b) By doing so, a quality surface is defined for this edge. (c) Curves depicting the quality values q along the original quality surface and the smoothed quality surface for a fixed y-value. By smoothing, sudden peaks are eradicated. (d) Vertices are sampled from the smoothed surface according to curvature criteria; the higher the curvature the more vertices are sampled. 63

2.16 (a) Example of adaptive sampling for i_{th} edge (highlighted) of a contour with 10 edges and two inner vertices. (b) Surface $S_{1,i}$ is defined by computing the minimum quality of each grid vertex taking into account $p_{I,1}$ as a second point. By smoothing $S_{1,i}$ and implementing the curvature loss criteria, vertices are sampled from the surface to form the set of inner vertices $V_{1,i}$. (c) In a similar fashion the surface $S_{2,i}$ is defined and vertices from it are sampled to form the set $V_{2,i}$. To train the NN, a pair of vertices with one vertex belonging to $V_{1,i}$ and another belonging to $V_{2,i}$ is sampled. The collected vertices must be at a distance of at least $0.3l_s$ from each other. 64

3.1 Histogram of contour populations with N_C edges that are used for training. The population of 6000 contours with 4 edges is found to be an adequate training set for acquiring satisfactory accuracy from the NNs involved in the meshing procedure. To retain or acquire a level accuracy needed for good quality meshing, for contours with 6 and 8 edges a training population of approximately 12000 and 24000 contours, respectively, is required; this leads to the choice of generating contour populations used for training that increase exponentially with the number of edges. 66

3.2 (a)-(g) In alphabetical order, the histogram of the contour populations for $N_C = (4,6,8,10,12,14,16)$ number of edges (Fig. 3.1) divided into groups according to the number of vertices N_I that are inserted by the reference mesher. Each of these groups is used to train NN_2 and NN_3. 67

3.3 The mean absolute error $\overline{e}$ as a function of target edge lengths l_s ranging from 0.2 to 1. The mean absolute error is defined as $\overline{e} = \overline{\left|N_I - \hat{N}_I\right|} = \sum_{i=1}^{n}(N_I^{(i)} - \hat{N}_I^{(i)})/n$, where $\hat{N}_I = (\hat{N}_I^{(1)},...,\hat{N}_I^{(n_{N_C})})$ are the number of vertices predicted by the NN, $N_I = (N_I^{(1)},...,N_I^{(n_{N_C})})$ are the number of vertices inserted from the reference mesher during refinement and n_{N_C} the number of contours with N_C edges in the test population. The mean absolute error $\overline{e}$ increases with increase in the number of contour edges N_C. For example, for a target edge length $l_s = 0.2$, $\overline{e}$ for the population of contours with 16 edges is approximately 2.3 times higher than the mean error for the population of contours with 4 edges due to the larger variation of number of inner vertices N_I that the reference mesher inserts for the populations of 16 edges. $\overline{e}$ also increases with the decrease of the target edge length l_s which is also due to the fact that the variation on the number of inner vertices N_I inserted by the reference mesher during refinement is larger for smaller target edge lengths. For instance, for the population of contours with 14 edges and the target edge lengths $l_s = 0.2$ and $l_s = 1$, $\overline{e}$ decreases from 1.3 for 0.2 respectively, while the standard deviation of inner vertices N_I for these target edge lengths decreases by 12%. 69

3.4 The real number of inner vertices N_I and the predicted number of inner vertices $\hat{N}_I$ as a function of the target edge length for a random contour with 12 edges (a) and a random contour of with 16 edges (b). For the contour with 12 edges the N_I and $\hat{N}_I$ differ by one point for $l_s = 0.2$, two points for $l_s = 0.3$ and are the same for the rest of the target edge lengths. For the contour with 16 edges, N_I and $\hat{N}_I$ differ by one point for the target edge lengths $l_s = \{0.2, 0.3, 0.4\}$ and are same for the rest of the values l_s. It can be concluded that NN_1 is appropriate to use for meshing purposes. 70

3.5 The mean squared error of euclidean distance $\overline{e}_{dist}$ as a function of grid resolutions of 10 × 10, 20 × 20, and 40 × 40. The mean square error is defined as $\overline{e}_{dist} = \overline{||p_{I,1} - \hat{p}_{I,1}||} = \sum_i^n (p_{I,1}^{(i)} - \hat{p}_{I,1}^{(i)})^2/n$, where $p_{I,1} = (p_{I,1}^{(1)},...,p_{I,1}^{(n)})$ are the real locations of the inner vertex inserted by the reference mesher during refinement and $\hat{p}_{I,1} = (\hat{p}_{I,1}^{(1)},...,\hat{p}_{I,1}^{(n)})$ are the predictions of the inner vertex for $n = 100$ contours with N_C edges. For every contour population, the error reaches convergence by using a grid with resolution 20 × 20. 71

3.6 (a)-(g): In alphabetical order, the average errors $\overline{e}_{worst} = \sum^{n_{N_{C,I}}} e_{worst}/n_{N_{C,I}}$ (%) and $\overline{e}_{mean} = \sum^{n_{N_{C,I}}} e_{mean}/n_{N_{C,I}}$ (%) for a number of $n_{N_{C,I}}$ random contours with $N_C = \{4, 6, 8, 10, 12, 14, 16\}$ edges as a function of inner vertices N_I. The range of inner points covers at least 68% of each contour population with N_C edges. Maximum $\overline{e}_{worst}$ of 23.41% occurs for the case of contours with 12 edges and 14 inner vertices (e). In most cases, both $\overline{e}_{worst}$ and $\overline{e}_{mean}$ tend to increase with the increase in the number of inner vertices; the displacement error from the predicted vertices of NN_2 increases with the number of vertices N_I , which, in turn, increases the triangulation errors $\overline{e}_{worst}$ and $\overline{e}_{mean}$. 72

3.7 (a) A contour with 8 edges with inner vertices inserted by the reference mesher during refinement (circular points) and inner vertices approximated by the scores of NN_2 (diamond points). (b) The graded reference mesh with its inner vertices. (c) The resulting mesh with approximated vertices. In this case, the triangulation errors account for $e_{worst} = 22.3\%$ and $e_{mean} = 9.8\%$. 73

3.8 (a)-(g): Validation of the developed triangulation algorithm. The average triangulation errors, $\overline{e}_{worst}$ and $\overline{e}_{mean}$, between meshes that are generated by using the triangulation algorithm with calculated connection table and the graded reference meshes for random contours with $N_C = \{4, 6, 8, 10, 12, 14, 16\}$ edges as a function of number of inner vertices N_I. The levels of triangulation errors indicate that the triangulation algorithm causes little to no significant error propagation in the connection scheme. 74

3.9 (a)-(g): Validation of the connectivity network NN_3. The average triangulation errors, $\overline{e}_{worst}$ and $\overline{e}_{mean}$, between contours meshed by using the triangulation algorithm (predicted connection table by NN_3) and the graded reference meshes for random contours with $N_C = \{4, 6, 8, 10, 12, 14, 16\}$ edges as a function of the number of inner vertices N_I. The inner vertices are those inserted in the cavity by the reference mesher during refinement. The accuracy of NN_3 is dependent on the number of N_I sampled inner vertices of a contour during the data augmentation process. For example in the case of contours with 6 edges for $N_I = 6$ (b) the training contour population is for 3% lower than that for $N_I = 4$ but there is reduction of 23% of $\overline{e}_{worst}$; this is due to the fact that the training of NN_3 for $N_I = 6$ relies on sampling 100 groups of six inner vertices for each contour, whereas for $N_I = 4$, 50 groups of four inner vertices are sampled. 75

3.10 Average worst triangulation error $\overline{e}_{worst}$ when NN_3^* is trained for different confidence levels of inner vertices populations (Table 3.1) with random and adaptive sampling. (a) For the prediction of the connectivity with one inner vertex at 96% confidence level, the sample population of inner vertices with adaptive sampling is 35% lower than that with random sampling. Even though NN_3^* is trained with lesser number of samples, 27% better accuracy is achieved. (b) For the prediction of the connectivity with two inner vertices at 90% confidence level, 18% better accuracy is achieved for a 22% lower sample population by applying adaptive sampling. (c) The accuracy for prediction of the connectivity with four inner vertices at 90% confidence level is 27% higher for a 17% lower sample population with adaptive sampling as compared to random sampling. 77

3.11 (a)-(g): The average triangulation errors, $\overline{e}_{worst}$ and $\overline{e}_{mean}$, between meshes generated with the meshing scheme and the graded reference meshes for random contours with $N_C = \{4, 6, 8, 10, 12, 14, 16\}$ edges as a function of number of inner vertices N_I. The inner vertices are those predicted by NN_2. In all cases, there is an increase of $\overline{e}_{worst}$ and $\overline{e}_{mean}$ compared to the previous tests where NN_2 and NN_3 are used separately. A maximum $\overline{e}_{worst}$ of 27.3% occurs for the case of contours with 16 edges and 18 inner vertices. 79

4.1 Example of mesh partitioning during reconnection algorithm for a square mesh. (a) Initially all elements that have a quality q_{el} lower than a quality threshold q_{thresh} (highlighted) are included in the set $E = \{1,2,3,4,5,6\}$. (b) The contours (dotted line) that are extracted after edge traversal of the elements $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ and $5 \rightarrow 6$ undergo the reconnection operation. (c) The triangulated contours after the applying the reconnection operation (Fig. 4.2). 84

4.2 From left to right: Example of a local mesh configuration containing elements below a quality threshold. Edges of the elements are deleted creating a contour cavity. The cavity undergoes the feature transformation F and is feeded to NN_3. NN_3 outputs values of the connection table. Based on the values of the connection table the triangulation algorithm of the meshing scheme meshes the contour cavity. The connectivity information of the elements is mapped back using F^{-1} to the original mesh to complete the reconnection process. 85

4.3 Example of mesh partitioning during the smoothing operation. A vertex v with $q_v \leq q_{thresh}$ is spotted (a). A new optimal vertex position is predicted using NN_S for the contour including all the elements that are connected with the vertex (dotted line) (b). 85

4.4 From left to right: Example of a local mesh configuration containing a vertex below a quality threshold. The elements that are connected to the vertex are deleted to create a contour cavity. The contour is transformed and the coordinates of the transformed contour are given as an input to NN_S. NN_S outputs the position of an inner vertex that is optimal quality-wise given that all contour points are connected with it. Finally, the vertex is mapped back to the mesh. . 86

4.5 Architecture of NN_S and NN_S^*. Both are feeding forward NNs that output the coordinates $\hat{p}_o = (x_o, y_o)$ of the optimal vertex position for the vertex repositioning and surface control operations. NN_S takes as input the contour coordinates P_C whereas NN_S^* takes as input P_C and also the tangents $t = (t_1, t_2)$ of the boundary or interface the reallocated vertex belongs to. 86

4.6 Example of local meshing configurations containing a low quality vertex that belongs to a boundary (a) and a low quality vertex that belongs to an interface (b). The edges of the element connected to the vertex are deleted to form an open contour in the case of a boundary vertex and a closed contour for the interface vertex. The contour coordinates along with tangents of the boundary or interface curve are transformed and given as an input to NN_S^*. NN_S^* outputs the optimal position of the vertex which is mapped back to the mesh. Finally, the vertex is projected to the curve. 88

4.7 Example of a local mesh configuration containing three long edges. First, a vertex is inserted in the middle of each long edge. The long edges are deleted to form a contour cavity containing the inserted vertices. The contour with the inner vertices are meshed using NN_3. Finally, the connectivity information is mapped through F^{-1} to the mesh. 89

4.8 Example of a local mesh configuration containing two short edges. First, a vertex is inserted in the middle of each short edge. The elements containing the short edges and their adjacent elements are deleted to form a contour cavity containing the inserted vertices. The contour with the inner vertices are meshed using NN_3. Finally, the connectivity information is mapped through F^{-1} to the mesh. 89

5.1 (a) The contour populations for training NN_3, NN_S and NN_S^*. The population of 6000 contours with 4 edges is found to be an adequate training set for acquiring good quality mesh improvement results. To retain the same level of accuracy the contour populations are increased at at a nearly exponential rate with the number of N_C edges. (b) The initial contour populations of $N_C = \{4, 5, 6, 7\}$ is augmented by sampling 10,20,30 and 50 N_I-pairs of inner vertices for $N_I = \{1, 2, 3, 4\}$ respectively. The training pairs (N_C, N_I) are used to train NN_3 for the application of the size control operations to mesh the contour cavities with the inner vertices that are created after partitioning the mesh. 92

5.2 Example of mesh improvement for a meshed square in a domain $\Omega = [0, 1] \times [0, 1]$. (a) Initially the mesh has a mean quality $q_{mean} = 0.97$, a minimum quality $q_{worst} = 0.83$, and a element angle distribution ranging between $43° \leq \theta \leq 94°$. (b) After random edge flipping and vertex perturbation the mean quality of the mesh decreases to $q_{mean} = 0.63$, the minimum quality decreases to $q_{worst} = 0.05$, and the element angles lie between $1° \leq \theta \leq 176°$. (c) After applying the reconnection and vertex repositioning operations, the mean quality of the mesh increases to $q_{mean} = 0.96$, the minimum quality $q_{worst} = 0.89$, and element angles lie between $41° \leq \theta \leq 84°$. 95

5.3 Example of applying the surface control operation to the interface vertices of a curve (highlighted) of a square mesh in a domain $\Omega = [0, 1] \times [0, 1]$. (a) Initially the mesh has a mean quality $q_{mean} = 0.93$, a minimum quality $q_{worst} = 0.77$, and a element angle distribution lying between $32° \leq \theta \leq 102°$. (b) After perturbing the interface vertices, the mean quality of the mesh decreases to $q_{mean} = 0.68$, the minimum quality decreases to $q_{worst} = 0.32$, and the element angles lie between $6° \leq \theta \leq 159°$. (c) After applying the surface control operation to the interface vertices, the mean quality of the mesh increases to $q_{mean} = 0.92$, the minimum quality increases to $q_{worst} = 0.7$, and the element angles range between $31° \leq \theta \leq 109°$. 96

5.4 Example of improving the quality of a square mesh in a domain $\Omega = [0,1] \times [0,1]$ containing a circular interface in the center (highlighted). (a) Initially the mesh has a mean quality $q_{mean} = 0.93$, a minimum quality $q_{worst} = 0.78$, and a element angle distribution ranging between $34^\circ \leq \theta \leq 96^\circ$. (b) After randomly flipping the edges and perturbing the vertices, the mean and minimum quality of the mesh decrease to $q_{mean} = 0.67$ and $q_{worst} = 0.04$, respectively, with element angles lying between $2^\circ \leq \theta \leq 175^\circ$. (c) After applying the reconnection operation followed by the vertex repositioning operation to the vertices to the perturbed mesh, the mean quality of the mesh increases to $q_{mean} = 0.92$, the minimum quality increases to $q_{worst} = 0.74$, and element angles lie between $29^\circ \leq \theta \leq 95^\circ$. 98

5.5 Example of applying the mesh improvement operations of a square mesh in a domain $\Omega = [0,1] \times [0,1]$ containing an airfoil shaped hole. (a) Initially the mesh has a mean quality $q_{mean} = 0.91$, a minimum quality $q_{worst} = 0.74$, and a element angle distribution lying between $32^\circ \leq \theta \leq 107^\circ$. (b) After perturbing the vertices and flipping the edges, the mean quality of the mesh decreases to $q_{mean} = 0.64$, the minimum quality decreases to $q_{worst} = 0.03$, and element angles lie between $7^\circ \leq \theta \leq 173^\circ$. (c) After applying the reconnection operation followed by vertex repositioning and surface control, the mean quality of the mesh increases to $q_{mean} = 0.94$, the minimum quality increases to $q_{worst} = 0.76$, and the element angles lie between $31^\circ \leq \theta \leq 102^\circ$. 100

5.6 (a)-(c): The circular interface's horizontal translation for different timesteps t. (d): The minimum quality q_{worst} as a function of the simulation timestep t. q_{worst} has a range between $0.50 \leq q_{worst} \leq 0.79$ before the application of the mesh improvement scheme and $0.69 \leq q_{worst} \leq 0.81$ after applying the scheme. (e) The minimum angles lie between $10 \leq \theta_{min} \leq 28$ before the application of the scheme and $28 \leq \theta_{min} \leq 40$ after the application. The maximum angles lie between $94 \leq \theta_{max} \leq 133$ before the application of the scheme and $80 \leq \theta_{max} \leq 108$ after the application. 102

5.7 (a)-(c): The collapsing circle for different timesteps. (d):The minimum quality q_{worst} as a function of the simulation timestep t. q_{worst} lies between $0.68 \leq q_{worst} \leq 0.74$ before the application of the scheme and $0.75 \leq q_{worst} \leq 0.79$ after the application. (e) The minimum angles lie between $27 \leq \theta_{min} \leq 38$ before the application of the scheme and $39 \leq \theta_{min} \leq 46$ after the application. The maximum angles lie between $108 \leq \theta_{max} \leq 116$ before the application of the scheme and $90 \leq \theta_{max} \leq 104$ after the application. 104

5.8 (a)-(c): Timesteps of the diagonal translation of the elliptical interface. (d): The minimum quality as a function of the simulation timestep. The minimum quality of the mesh q_{worst} lies between $0.41 \leq q_{worst} \leq 0.77$ before the application of the mesh improvement scheme and $0.69 \leq q_{worst} \leq 0.83$ after its application. (e) The minimum angles lie between $19 \leq \theta_{min} \leq 36$ before the application of the scheme and $33 \leq \theta_{min} \leq 45$ after the application. The maximum angles lie between $94 \leq \theta_{max} \leq 133$ before the application of the scheme and $84 \leq \theta_{max} \leq 100$ after the application. 105

5.9 (a)-(c): Timesteps of the zalesak disc rotation. (d): The initial interface and the deformed interface after the completion of the rotation. (e): The minimum quality as a function of the simulation timestep. q_{worst} lies between $0.58 \leq q_{worst} \leq 0.69$ before applying the mesh improvement scheme and $0.68 \leq q_{worst} \leq 0.74$ after its application. (f) The minimum angles lie between $26 \leq \theta_{min} \leq 32$ before the application of the scheme and $35 \leq \theta_{min} \leq 44$ after the application. The maximum angles lie between $99 \leq \theta_{max} \leq 117$ before the application of the scheme and $90 \leq \theta_{max} \leq 107$ after the application. 108

5.10 (a)-(c): Timesteps of the vortex flow deformation of the circular interface. (d): The initial interface and the deformed interface after the completion of the simulation. (e): The minimum quality as a function of the simulation timestep. q_{worst} has a range between $0.58 \leq q_{worst} \leq 0.79$ before the application of the mesh improvement scheme and $0.71 \leq q_{worst} \leq 0.84$ after the application. (f) The minimum angles lie between $21 \leq \theta_{min} \leq 32$ before the application of the scheme and $27 \leq \theta_{min} \leq 37$ after the application. The maximum angles lie between $98 \leq \theta_{max} \leq 120$ before the application of the scheme and $90 \leq \theta_{max} \leq 108$ after the application. 110

6.1 Example of the refinement process using NN_3 to create a mesh with a target edge length l_s for a high resolution contour S (150 edges) forming a circle. (a) Points are sampled from S to form an initial contour. (b) NN_3 is called to mesh the initial contour. (c) If an edge of the initial mesh has a length l_e that is bigger than l_s, then $n_K = K - 1$ vertices are inserted to the edge, where $K = [l_e / l_s]$. (d) If the vertices of an edge belong to the high resolution contour S, the inserted vertices are projected to S. NN_1 and NN_2 are used to predict the number and location of inner vertices for each sub-contour with a target edge length equal to the average edge length of each sub-contour. (e) Each sub-contour composed of vertices of the elements, the inserted vertices, and the predicted inner points is meshed using NN_3. As a post-treatment, the vertex repositioning operations are applied by calling NN_S and NN_S^*. (f) The refinement process is repeated until the edge lengths are close to l_s. 114

6.2 Example of adaptive meshing for a high resolution contour S (201 edges) forming an airfoil. (a) A sizing function h is defined over the inner domain Ω_S. The values of the sizing function represent the local target edge lengths that will dictate the elements sizes. The darker areas represent smaller values of the sizing function, i.e regions where smaller elements should be created to better approximate the geometry of the airfoil. (b) Points are sampled from S to form an initial contour that is meshed using NN_3 (c). Points are inserted incrementally on each edge until all lengths of the segments that are created after the subdivision are close to the assigned size function value for each inserted vertex. The number of interior points and their location are predicted using NN_1 and NN_2. (e) Finally, after meshing each sub-contour with its inner vertices using NN_3, NN_S and NN_S^* are called to improve the quality. 116

6.3 Example of adaptive meshing for a high resolution contour with a circular hole in the middle.(a) The sizing function is defined such that elements of smaller size are created near the circular hole . (b) Since NN_3 is able to only mesh contours that are watertight (i.e no holes), the high resolution contour is divided into four sub-regions (contours containing 150 edges). Points are sampled from the sub-regions to form four contours. (c) Each of the four contours, is meshed using NN_3. (d) Based on the initial meshes, the adaptive meshing process is applied and iterated until the edge lengths of elements are close to the values of the assigned sizing function. 118

A.1 Example of generating a graded mesh of a contour geometry with $Gmsh^{©}$. (a) In the $input.geo$ file the points, edges (*Segments*), contour (*LineLoop*) define the geometry input. The *Trasfinite* command constraints vertices from being inserted at the edges of the contour. *Mesh.CharacteristicLengthFactor* defines the uniform target edges length. (b) The geometry of the $input.geo$ file. (c) The graded mesh after calling the command "$input.geo -2 -algo\ del2d\ output.msh$". 138

A.2 Example of generating a graded mesh of the zalesak disc geometry with $Gmsh^{©}$. (a) In the $input.geo$ file first the geometry of the square border is created and its points are assigned a target edge length 0.5. Next, the geometry of the circle and the slot is created. The slotted disc is a result of a boolean difference btw the surface of the circle and the slot. The vertices of the slotted disc interface are assigned a target edge length 0.1. The size of the mesh elements will then be computed by interpolating these values of the square's points and the interface's points (Mesh.MeshSizeFromPoints). (b) The geometry of the $input.geo$ file. (c) The graded mesh of zalesak disc after calling the command "$input.geo\ -2\ -algo\ del2d\ output.msh$". 140

A.3 Example of generating a graded mesh of the square with the parabolic interface geometry with $Gmsh^{©}$. (a) In the $input.geo$ file first the geometry of the square border are created and its assigned a target edge length 0.5. Next, the geometry parabolic interface is created. The slotted disc is a result of a boolean difference btw the surface of the circle and the slot. The vertices of the parabolic interface are assigned a target edge length 0.1. The size of the mesh elements will then be computed by interpolating these values of the square's points and the interface's points (Mesh.MeshSizeFromPoints). (b) The geometry of the $input.geo$ file. (c) The graded mesh of the square including the parabolic interface after calling the command "$input.geo -2 -algo\ del2d\ output.msh$". 141

A.4 (a) Example of a locked contour vertex p_1. Starting from the contour facet $\{p_0, p_1\}$ of element E_1 the adjacent by edges of the elements E_2 and E_3 are visited via the traversal of facets $\{p_0, p_1\} \rightarrow \{p_1, p_{I,1}\} \rightarrow \{p_1, p_{I,2}\} \rightarrow \{p_1, p_2\}$. The facet $\{p_1, p_2\}$ is a contour facet. Therefore, the contour vertex p_1 is locked. (b) Exampled of a locked interior vertex. Edges of elements surrounding the interior vertex p_I are traversed. Starting from facet $\{p_I, p_0\}$ this leads to the edge traversal $\{p_I, p_0\} \rightarrow \{p_I, p_1\} \rightarrow \{p_I, p_2\} \rightarrow \{p_I, p_3\} \rightarrow \{p_I, p_4\} \rightarrow \{p_I, p_5\} \rightarrow \{p_I, p_6\} \rightarrow \{p_I, p_0\}$. The starting facet $\{p_I, p_0\}$ is also the last visited facet. Therefore, p_I is locked. 142

A.5 Example of spotting a sub-contour P. After the connection of of contour facets with element. The vertices (contour or inner) $\{p_0, p_1, p_2, p_3, p_4, p_5, p_6\}$ are open for further connections and are contained in the set of vertices to visit V_{visit}. The list of facets to visit F_{visit} are the facets that link the open vertices. F_{visit} contains facets that are located in the interior of a contour (appearing after the creation of elements) or contour facets. Starting from p_0, $v_{next} = p_1$ is visited through the facet of $F_{visit} = \{p_0, p_1\}$. p_1 is removed from the list V_{visit}. The rest of the vertices from V_{visit} are visited using the facets of F_{visit} in a similar fashion. Each time a vertex is visited it is removed from the set V_{visit}. The traversal stops at p_0 which is the initial visiting vertex. All the edges visited from F_{visit} are contained the the set $F_{contour}$ which contains the facets of sub-contour P. . . . 143

List of Tables

1.1 Examples of compound operations composed by the successive application of other operations. 19

3.1 Sampling sizes for the mean populations of the contours inner vertices with random and adaptive sampling for different confidence levels. 78

6.1 The number of elements N_{el}, minimum angles θ_{min}, maximum angles θ_{max}, worst quality q_{worst} and the mean quality q_{mean} for the circle (Fig. 6.1), airfoil (Fig. 6.2) and contour with circular hole (Fig. 6.3) examples (the closer a quality value is to 1 the better). 118

7.1 Weekly growth rate (%) of 3D contour training datasets along with the intended initial goal population. 124

Nomenclature

List of Abbreviations

AI	Artificial Intelligence
ALE	Arbitrary Lagrangian Eulerian
BMU	Best Matching Unit
CAD	Computer Aided Design
CDT	Constrained Delaunay Triangulation
CNN	Convolutional Neural Networks
FEM	Finite Element Method
GCNN	Geodesic Convolutional Neural Networks
GCN	Graph Convolutional Networks
GNG	Growing Neural Gas
LEPP	Longest Edge Propagation Path
LIG	Let It Grow
NN	Artificial Neural Network
PDE	Partial Differential equations
ReLU	Rectified Linear Unit
RNN	Recursive Neural Networks
SBMU	Second Best Matching Unit
SGD	Stochastic Gradient Descent
SOM	Self Organizing Maps
SPR	Small Polyhedron Reconstruction

Greek Letters

Γ	Discretized geometry domain
λ	Weight decay
Ω	Geometry domain
ω	Angular velocity
Ω_s	High resolution domain contour
θ	Angle
θ_{max}	Maximum angle
θ_{min}	Minimum angle

Latin Letters

ΔT	Period
ℓ	Loss function per training sample
$\hat{A}$	Estimated connection table
$\hat{N}_I$	Estimated number of inner vertices
$\hat{p}_I$	Estimated inner vertices coordinates
$\hat{p}_o$	Estimated optimal vertex position
$\hat{r}$	Unit vectors along radius r
$\hat{S}_{G_k}$	Estimated score of grid points included in a patch
$\hat{s}_{i,j}$	Estimated score of grid point
$\hat{x}, \hat{y}$	Unit vectors along x and y axis
$\mathscr{A}$	Convolution matrix
$\mathscr{F}_K$	Stride step size
$\mathscr{L}$	Number of hidden layers
$\overline{e}$	Mean absolute error of NN_1 prediction
∂V	boundary of domain (contour)
v	Velocity

A	Connection table
a	Neural Network free parameters
$A_{ordered}$	Ordered connection table
b	Neuron bias
C	Polyhedral cavity
c	Number of hidden layers
d_A	Dimension of connection table
E	List of elements
e_{mean}	Relative difference of q_{mean} between predicted mesh and reference mesh
e_{worst}	Relative difference of q_{worst} between predicted mesh and reference mesh
F	Procrustes superposition transform
F^{-1}	Inverse procrustes superposition transformation
G	Grid
g	Activation function
G_k	Grid patch
h	Sizing function
I, J	Set of tetrahedra
L	Hidden layers
l_e	Edge length
l_s	Target edge length
L_{thresh}	Long edge length threshold
l_{thresh}	Short edge length threshold
M	Pooling matrix
m	Momentum term
$N_I^{'}$	Sub-contour number of inner vertices
n_{batch}	Number of trainind batch included in a batch
N_C	Number of contour edges

Nomenclature

N_{el}	Number of elements
N_{G_k}	Number of grid points included in a patch
N_G	Grid resolution
N_I	Number of inner vertices
$n_{N_{C,I}}$	Number of contours with N_C edges and N_I inner vertices
n_{N_C}	Number of contours with N_C edges
N_p	Number of patches
N_{train}	Number of traning population
NN_3^*	Neural network used for the prediction of the connectivity with adaptive point strategy
NN_S^*	Neural Network used for the application of surface control
NN_1	Neural network used for the prediction of the number of inner vertices
NN_2	Neural network used for the prediction of the location of inner vertices
NN_3	Neural network used for the prediction of the connectivity
NN_S	Neural Network used for the application of vertex repositioning
P	Set of vertices
$P_C^{'}$	Sub-contour vertices coordinates
$P_I^{'}$	Sub-contour inner vertices coordinates
P_C	Contour vertices coordinates
P_{G_k}	Grid point coordinates included in a patch
P_I	Inner vertices coordinates
p_I	Inner vertices coordinates
p_o	optimal vertex position
Q_C	Reference contour vertices coordinates
q_{el}	Quality of element
q_{mean}	Mean quality value of elements in mesh
q_v	Quality of vertex
q_{worst}	Minimum quality value of element in mesh

R	Domain of inner vertices ring
r	Radius
S	Quality surface
S_{G_k}	Score of grid points included in a patch
S_G	Scores of grid points
$s_{i,j}$	Score of grid point
T	Total number of timesteps
t	Tangents
u	Combination function
V	Interior domain (cavity)
w	Synaptic weights
X	Training input batch
x, y	Cartesian coordinates
Y	Labels of training input batch
e_{dist}	Squared error btw location of original vertex location and approximated one

1 Introduction

1.1 Motivation

Mesh generation concerns the tessellation of geometry's domain into a set of elements. The shape of the elements varies from triangles and quadrilaterals in 2D to tetrahedra, pyramids, prisms, hexahedra and other polyhedral elements in 3D. Triangle and tetrahedral elements are also known as simplicial elements. Meshes are used in multiple scientific fields. The discretization of Partial Differential Equations (PDE) is based on the use of meshes to approximate solutions of underlying physics in fields such as structural analysis (Portaneri et al. (2019), Sumner & Popović (2004)), fluid dynamics (Baker (1997), Lohner (1995), Liang et al. (2007)) , aerodynamics (Hassan et al. (1996)), quantum mechanics (Solanpää & Räsänen (2018)) etc. In Computer Aided Design (CAD), meshes are used for the tessellation of stereolithographic files (Beniere et al. (2013), Lavoué et al. (2005)) and surface reconstruction from point clouds (Marton et al. (2009)). In computer graphics, meshes are utilized to render objects, for animation and visual effects (Portaneri et al. (2019), Sumner & Popović (2004)).

For the tessellation of complex geometries, adaptivity is an essential mesh property. Physical objects are usually simple and smooth in some parts, complex and contorted in others. Therefore, elements of smaller size are needed to accurately approximate complex regions of a geometry. However, limited computational resources can restrict the generation of a uniform mesh with elements with the same size as the ones used to approximate the complex regions. Thus, small elements and bigger ones must co-exist in a mesh. Mesh generation algorithms must also satisfy element shape requirements. The aforementioned applications can either require a mesh with isotropic elements that approach the shape of their regular polygon counter part or anisotropic elements that are elongated in a particular direction. These requirements lead to the development of mesh generation algorithms that can suffer from complex code of extensive size, explicit handling and a significant computational cost. Meshing algorithms are difficult to be transported to acceleration platforms such as GPU architectures limiting the choices of computational frameworks. Moreover, mesh generation algorithms do not always guarantee a mesh that meets the element shape and size qualifications. To ensure efficiency, post mesh improvement heuristics that move the vertices of the elements and re-adjust their connectivity might be required. Mesh improvement can either be applied globally over the whole domain of the mesh or locally to regions of badly shaped elements. Local mesh improvement is opted for meshes used in simulations where the vertices move according to the motion of the material points. Local mesh improvement has to be applied at each simulation step to ensure a converging solution. The heuristic nature of local mesh improvement algorithms does not always guarantee robustness and could account for a significant computational cost over the course of a simulation.

There is therefore a demand for a computationally efficient meshing and mesh improvement framework that is able to comply to the needs and exceptions of a geometry, satisfy mesh requirements, and avoids as much as possible explicit treatment. Machine learning algorithms use computational methods that are able to solve complex problems based on data observation and pattern recognition without relying on predetermined equations or explicit algorithm implementation. Machine learning has been successfully applied for the resolution of complex tasks such as image pattern recognition (Egmont-Petersen et al. (2002)), natural language processing (Young et al. (2018)), autonomous driving (Grigorescu et al. (2020)) etc. Therefore, machine learning has the potential to be a useful tool for mesh generation and mesh improvement. Existing meshing algorithms are able to provide datasets to train machine learning models for mesh generation and mesh improvement without the need of the underlying meshing technique involved. This data driven approach has the potential to provide an automated meshing framework that bypasses the complexities and computational hurdles of existing meshing algorithms. It is, therefore, within the scope of the present research work to simplify the process of meshing and to adapt mesh generation and mesh improvement to an automatic data driven framework using machine learning tools that could help overcome the aforementioned issues.

1.2 Meshes

The generation of a mesh starts with an input geometry or a set of vertices (point cloud) to be tessellated embedded in a 2D or a 3D dimensional space. For a given geometry, the tessellation can either take part on the boundary of the geometry (surface meshing) or its inner domain. The boundary of a geometry can either have an explicit representation of parametric surface or b-spline surface, or an explicit representation of a lower dimension mesh (e.g poly line representation for 2D mesh generation, triangular face representation for 3D mesh generation). To respect the features of the target geometry and provide an optimal discrete approximation its boundary, elements have to be of appropriate size with respect to more complex regions of interest (e.g curvature of geometry).

With respect to numerical computation, the shape and size of elements in a mesh play a crucial role to the behavior of numerical methods that are based on the discretization of partial differential equations (e.g. Finite Element Method (FEM), Finite Volume Method (FVM)). For simplicial elements, the shape of triangular elements can be characterized in terms of angles of adjacent edges. For tetrahedral elements, the shape is characterized in terms of dihedral angles between adjacent triangular faces and the plane angles between the adjacent edges on each triangular face. Badly shaped elements introduce errors in the interpolation of field values (in particular on the derivatives of basis functions), on the condition number of the stiffness matrix used to solve the linear system of the involved numerical method and on the approximation of the field values with respect to the asymptotic solution (Shewchuk (2002c)). In Cheng et al. (2012), the following observations are made with respect to the effects of the element shape:

- Triangular elements with large angles (cap) (Fig.1.1a) and tetrahedral elements with large dihedral angles (sliver) (Fig.1.1b) introduce large errors in numerical approximations of differential operators. The errors in the gradients becomes unbounded as triangle angles and dihedral angles approach 180°.
- Triangular elements with small angles and tetrahedral elements with small dihedral angles lead to poorly conditioned numerical integration schemes that compromise the condition number of the stiffness matrix.
- Highly skewed elements occupying small areas (2D) or volume (3D) place restrictions on the maximum time-step when solving time dependent problems using explicit integration techniques.

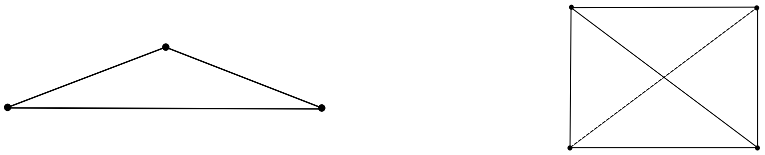

(a) (b)

Figure 1.1: A 2D triangular cap element with a large angle (a) and a 3D sliver tetrahedral element (b) with large dihedral angles. These type of elements introduce large errors to the gradients of the basis functions.

The size and shape of an element can be encapsulated as a numerical value known as quality. Multiple quality measures of simplicial elements have been proposed based on geometrical features of the element; these include measures based on the minimum sine of the element (Freitag & Ollivier-Gooch (1997a)), the area to edge length and volume to edge length ratio for 2D and 3D simplicial elements respectively (Parthasarathy et al. (1994)), the radius ratio between the inscribed and circumscribed hypersphere (Caendish et al. (1985)) and so on. Mesh improvement algorithms aim at improving the quality of the elements of a mesh.

Simulations of material motion (e.g deformation, fluid flow) modeled by continuum mechanics make use of algorithms solving the underlying partial differential equations (PDE) that follow either a Eulerian description, a Lagrangian description (Malvern (1969)) or an arbitrary Lagragian-Eulerian (ALE) description (Donea et al. (1982)). In the Eulerian description, which is widely used in fluid mechanics, the computational mesh is fixed and the material points can move through it. In the Lagrangian description, which is mostly used to simulate elastic and plastic solids, the vertices of the mesh move in accordance with the motion of the material points. The vertices of a mesh following the ALE description can either be held fixed in a Eulerian manner or moved in a Lagrangian fashion. Eulerian algorithms can handle large distortions of the material at an expense of progressive smoothing (blur) of the field values (velocity, deformation gradient, phase field) caused by the projection of the material points to the mesh. The smoothing of the field values produces numerical errors such as artificial viscosity, artificial plasticity, or the disappearance of small features (e.g thin sheet artificial evaporation, smoothing of surface details). On the other hand, Lagrangian algorithms allow for lower smoothing of the field values, an easy handling of free surfaces and interfaces between different materials but is unable to follow large distortions without recoursing to mesh improvement techniques. Global mesh improvement is avoided since it can be computationally costly, and can quickly accumulate large numerical errors because of the need to re-interpolate physical properties such as velocity and strain from the old mesh to the new one. Hence, local mesh improvement is favored. Local mesh improvement algorithms rely on operations that improve the mesh quality by relocating the position of a vertex to improve the quality of the adjoint elements and/or by changing the mesh topology locally. Usually such a local approach in mesh improvement entails the use of a scheme where different combinations of operations are applied heuristically that improve the badly shaped elements of the mesh.

With regards to their underlying topology, meshes can be categorized into structured and unstructured meshes. Structured meshes are characterized by regular configurations in which all vertices and elements are represented in form of a uniform template. Structured meshes support implicit connectivity with vertices and faces usually being aligned with the coordinate axis (Fig. 1.2a). Simplicial structured meshes can either be trivially constructed by block subdivision of a regular grid (Allwright (1988)), be algebraic (Cook (1974)), or rely on solving PDE equations over the domain to be tessellated (Thompson et al. (1985)). Due to the regularity in indexing and connectivity, structured meshes offer a low computational storage and can facilitate the application of efficient numerical methods. However, due to topological regularity, problems can occur when complex geometry constraints are met or if variable spatial resolution is desired. Such constraints, limit the use of structured meshes for the tessellation of simple geometrical domains.

On the other hand, unstructured meshes do not have regular connectivities instead such meshes tesselate a domain into a set of irregular triangle or tetrahedral elements (Fig. 1.2b). Compared to structured meshes, unstructured ones offer a far more geometric flexibility, variable spatial resolution and are generally preferred when tessellating complex objects. Although the development of unstructured mesh generation algorithm entails a greater complexity and

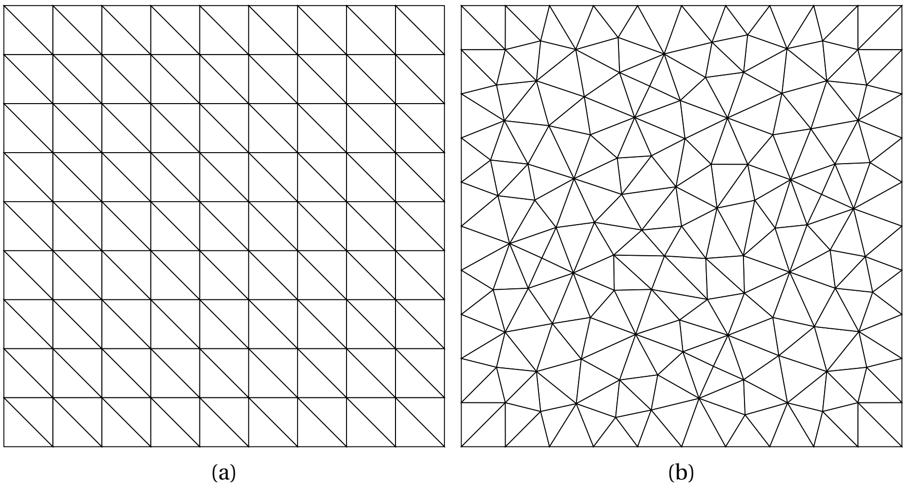

(a)
(b)

Figure 1.2: Examples of a structured mesh (a) and an unstructured mesh (b).

computational cost compared to structured mesh generation algorithms the potential for variable and adaptive spatial resolution can lead to significant computational savings due to a reduced total element count. The mesh generation algorithms for the creation of unstructured meshes can be separated into the following categories:

1. Grid based methods (Quadtree/Octree)
2. Advancing front
3. Delaunay methods
4. Hybrid methods

In what follows, a brief description of the aforementioned unstructured mesh generation and mesh improvement methods is provided.

1.3 Mesh Generation algorithms

1.3.1 Quadtree/Octree

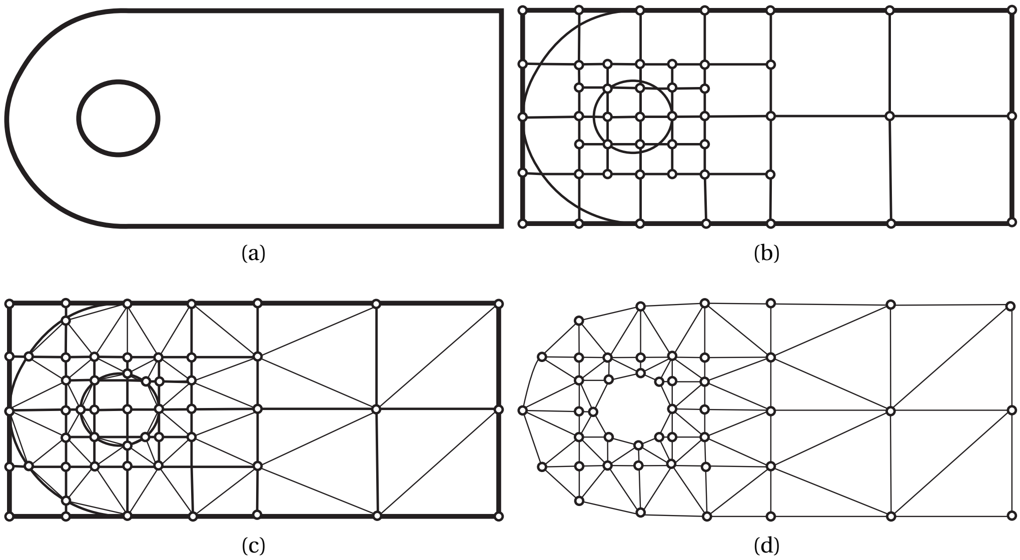


Figure 1.3: (a) The original geometry (b) A bounding box occludes the geometry and cells are inserted. Smaller cells are inserted for a better approximation of more complex regions. (c) The cells are meshed according to templates. (d) The final mesh after snapping the vertices of boundary cells to vertices of the geometry's boundary and cutting elements with vertices outside the geometry domain.

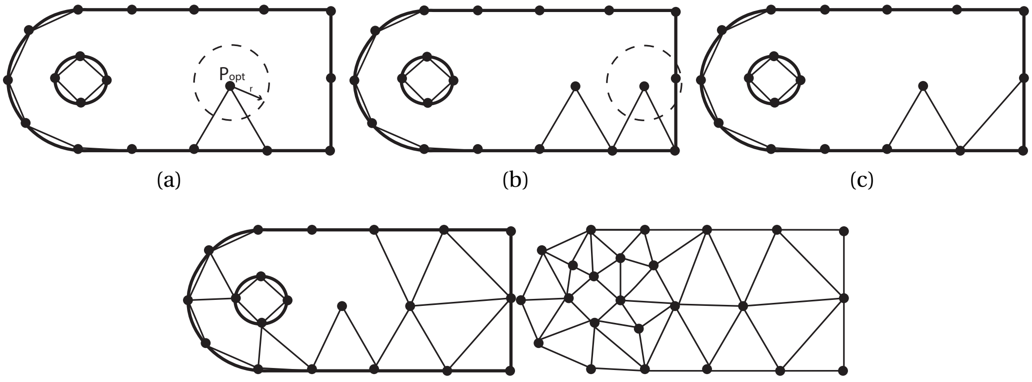
Quadtree and Octree methods (Yerry & Shephard (1983), Schneiders & Bünten (1995), Scheiders (2000), Greaves & Borthwick (1999), Fischer & Bar-Yoseph (2000), Maréchal (2001)) rely on the use of a grid that covers the boundary of a geometry to generate triangular and tetrahedral meshes, respectively. The structure of quadtrees and octrees is a tree data structure that can be applied to subdivide the dimensional space of the geometry by means of recursive subdivision with cells known as quads (2D) and octants (3D). Initially, a bounding box that encloses the geometry is first created and then is filled with four equally sized cells. Each cell is then subdivided several times to produce cells of smaller size (Fig. 1.3b). Stopping criterions for the subdivision could be based on the local geometry of a domain (e.g local curvature of a boundary), the distance of point cells to the surface/curve, a prescribed element size function, a velocity and deformation gradient, a maximum level of refinement etc. Defined rules guarantee the termination of the subdivision process, such a rule is the one level difference (2:1 rule); the rule states that two cells sharing at least an edge are at the same or subsequent depth of the tree structure.

After the creation of the cells in the bounding box the simplicial elements are generated (Fig. 1.3c). The cells corner vertices are considered to be part of the mesh vertices. The simplices are created by subdividing the cells according to predefined patterns corresponding to the possible cell vertices configuration. For a cell, the application of the 2:1 rule reduces the amount of possible internal vertices configurations with regards to its neighbor cells. The are 16 possible patterns for the creation of triangular elements and 78 for the creation of tetrahedral elements. To reduce the number of patterns for the creation of tetrahedral elements, in 6 patterns are used after observing that those patterns represent 90% of internal oct-cells (Yerry & Shephard (1983)). The rest of the octants are meshed using a fast tetrahedralization algorithm where vertices of the octant are connected with an interior vertex. Another approach applies Delaunay criterion (Schroeder & Shephard (1990)) (see Section 1.3.3) to create tetrahedral elements in an oct-cell. To create compatible triangulations between the overlapping faces of the oct-cells this procedure is followed in an orderly manner. An alternative way to create tetrahedral elements involves inserting at a first step a vertex at the center of each oct cell and connect the vertices of each of its face with it to form tetrahedral elements (Frey & George (2007)). Next, the faces of the oct-cells are triangulated using the 16 predefined patterns and each triangular face is connected with the centroid vertex.

After generating the simplices a mesh is created for the bounding box that encloses the boundary of the geometry. The last step of the algorithm involves conforming the mesh to the boundary of the geometry (Fig. 1.3d). To recover the mesh of the geometry, the vertices of the cells are categorized through a coloring scheme as being outside or inside the geometry. Cells with all vertices being outside the geometry are discarded. Cutting points are introduced to each boundary cell edge with vertices that are located outside and inside the geometry. After cutting the cells according to the cutting points and projecting boundary vertices to geometry boundary, further subdivision may be needed to the resulted boundary faces for the introduction of new boundary simplices (Labelle & Shewchuk (2007)). Another approach relies on further refining tetrahedra located in boundary cells and deforming them through an optimization procedure to adapt the boundary of the geometry (Neil Molino & Fedkiw (2003)). Although these strategies can be effective with respect to boundary conformity, they may produce badly shaped elements near the boundary or lead to a mesh that is much denser in elements near the boundary than the interior of the domain.

Tree data structures are very efficient in storing and querying geometrical information. Octree and quadtree mesh generation algorithms are very fast and robust providing elements of an appropriate shape and size in the interior of the geometry. A consequence of the nature of the algorithm is that the worst shaped elements are located at the boundary of the geometry. Another drawback of these types of algorithm is the difficulty to handle mesh sizing. The application of rules to the nodes of the tree, such as the one level difference limits the choices of cell sizing (power of 2) and results to a limited mesh density gradation as well. This could be solved by applying more sophisticated rules to the cells but, as a trade off, the search of proper predefined patterns to create elements of appropriate shape becomes more complicated. As a consequence, octrees and quadtrees are less suitable for adaptivity.

1.3.2 Advancing front



(a) (b) (c)

(d) (e)

Figure 1.4: (a) The boundary of the geometry is discretized according to a user defined edge length. An edge (facet) is connected to a vertex P_{opt} to form an element. A circle of radius r centered at P_{opt} is used to spot possible intersections with other fronts or to check if other vertices are included in it. (b) A vertex of another front is spotted inside the circle of the candidate vertex. (c) In this case the original candidate vertex is rejected and the vertex that belongs to the neighbor front is selected to form a new element. (d) The formation of the front after several stages of the method. (e) The final mesh.

The advancing front method (George (1971), Seveno et al. (1997), Löhner & Parikh (1988), Kallinderis et al. (1995), Löhner (1996)) starts with the discretization of the geometry boundary into facets, i.e edges in 2D and triangular faces in 3D, with a length that meets user defined criteria. These facets form the initial front which advances into the interior domain of the geometry. At each step, a facet is selected and is connected with a vertex in the interior domain forming an element. After the creation of the element, the facet is removed from the front, leading to a new front. This process iterates until all fronts are merged and the interior domain is covered by elements (Fig. 1.4).

A critical feature of advancing front algorithms is the choice of vertex to connect the facet with. The vertex may already exist or a new one must be chosen to form an element of optimal shape and size. The newly formed element might intersect with other fronts and thus is rejected. Additionally, a candidate vertex might be located close to a front vertex. In this case the front vertex is preferred as a new candidate vertex for element formation instead to avoid the formation of element with small edges at some later stage.

The method starts with a selection of a facet based on some criteria, such as minimum edge length. The next stage involves the selection of a vertex to connect it with the facet to form an element. A quite common strategy of vertex selection for a facet is to start with the placement of an optimal vertex P_{opt} (Fig. 1.4a). P_{opt} is selected so that the tentative element formed with the connection to the facet satisfies element size and shape. As a next stage, a spatial search for a potential alternative candidate vertex P is conducted. This involves the search within a hypersphere around P_{opt} (circle in 2D, sphere in 3D) with a radius size related to regional element size requirements. If no new vertices are located inside the hypersphere an element is formed using P_{opt} as a connecting vertex with the facet. If vertices are located inside the hypersphere they are ordered with respect to the increasing distance from P_{opt} (Fig. 1.4b, Fig. 1.4c). Subsequently, the formed element is checked for potential intersections. If the elements formed with all candidate vertices P lead to intersections, then the process of optimal point placement is repeated with reduced element size.

Unlike grid based methods, advancing front methods comply with the boundary discretization and provide elements of appropriate shape in the boundary layer. Another offered advantage over grid based methods, is that they are invariant with respect to rigid motions of the geometry. A main issue with advance front methods is that they present problems when fronts merge in regions where there are sudden changes of element size requirements; the procedure might need to restart with new parameters (Seveno et al. (1997)). As a result, the convergence of the method in 3D is not guaranteed. Moreover, the intersection checking phase renders the meshing process rather slow when compared to other methods.

1.3.3 Delaunay methods

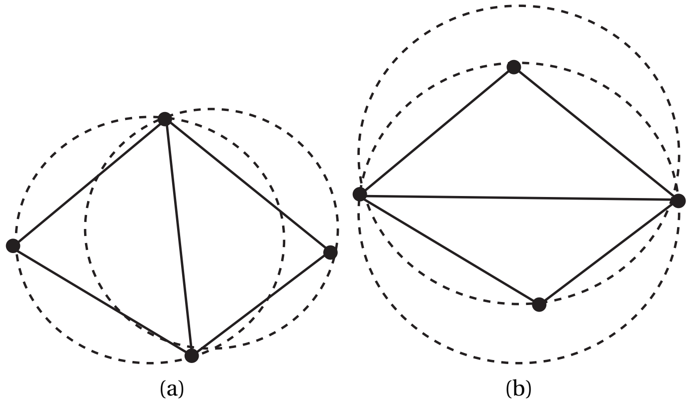

(a)
(b)

Figure 1.5: Example of Delaunay criterion. (a) maintains the criterion while (b) does not.

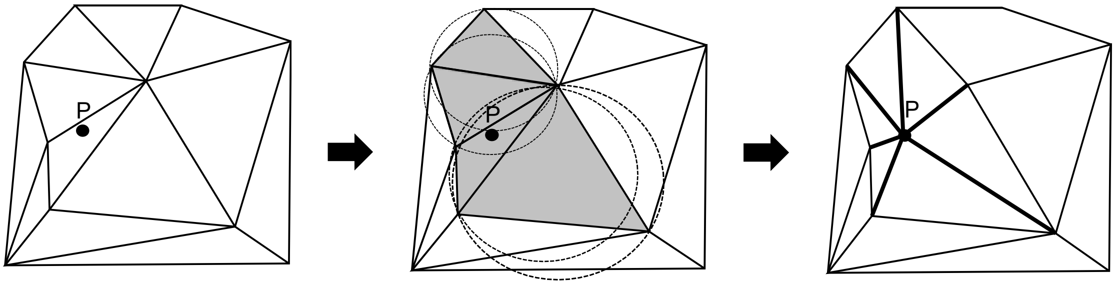


Figure 1.6: From left to right: A vertex *P* is inserted in a triangulation The vertex is included into the circumcircle of the shaded triangular elements The elements whose circumcircle include *P* are deleted and the vertices of the countour cavity formed after the deletion are connected with *P*.

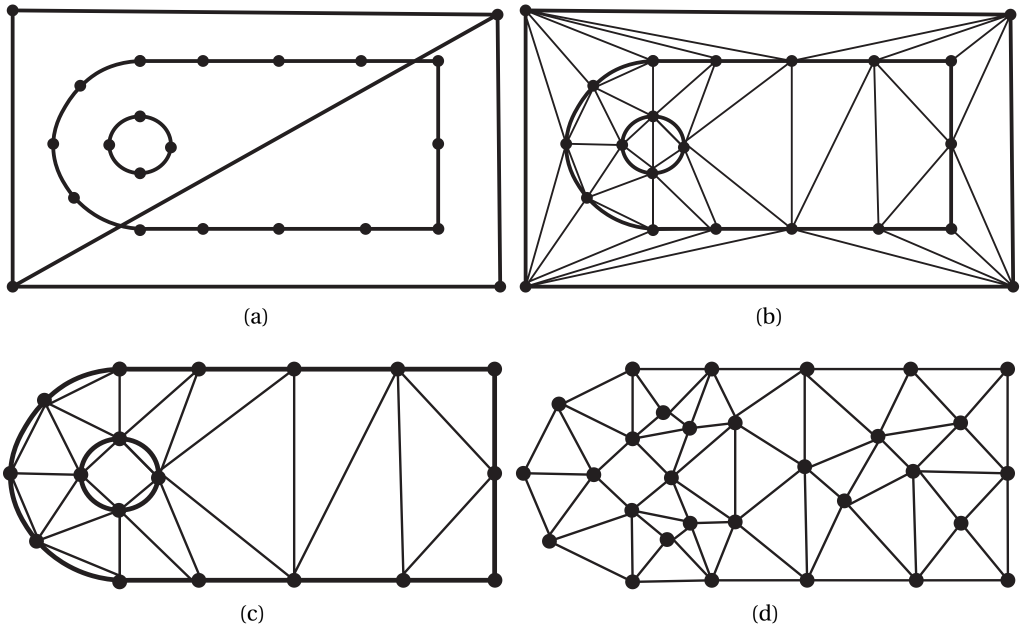


Figure 1.7: Steps of the CDT algorithm. (a) A bounding box composed of two simplices (triangles) occludes the vertices of the discretized boundary of the geometry. (b) An initial triangulation is performed using the Boywer-Watson algorithm. The initial triangulation contains the vertices of the discretized boundary and the vertices of the boundary box. (c) After the initial triangulation, the boundary facets are recovered and the elements that are located outside of the geometry are deleted (d) Delaunay refinement inserts new vertices to strategic locations of the initial triangulation to improve the size and shape of the elements.

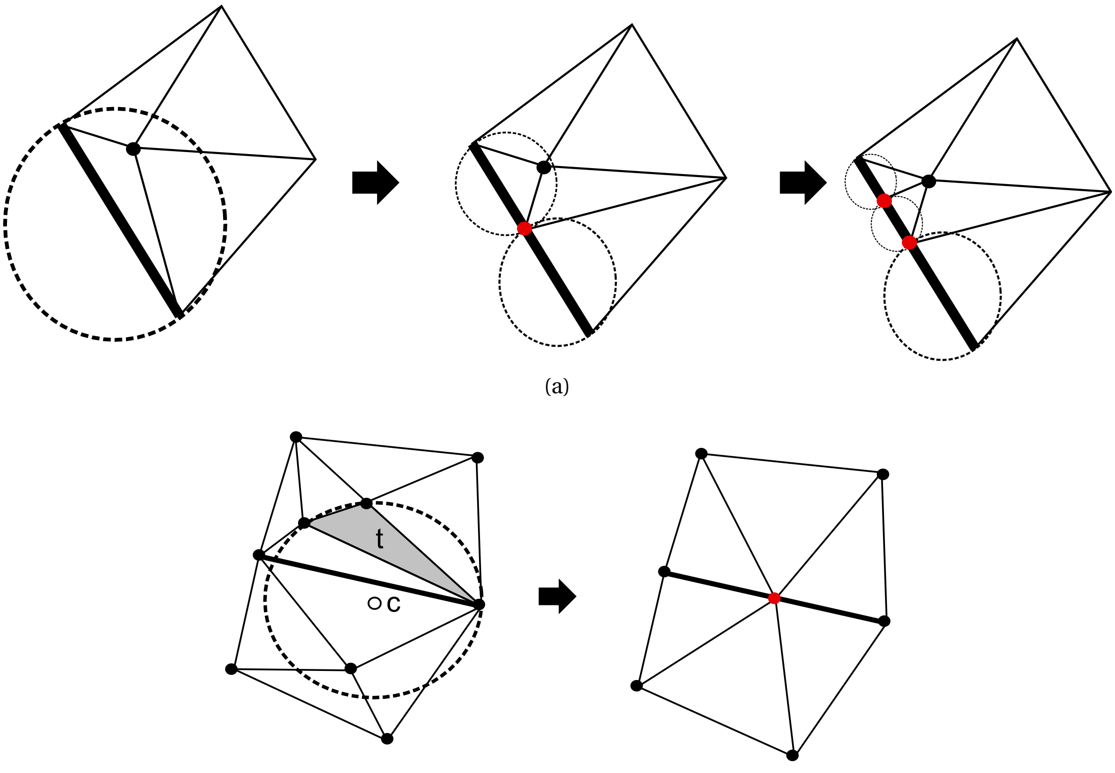
Delaunay mesh algorithms aim at creating elements whose circumscribed hypersphere does not contain any other vertex (Fig. 1.5). The Delaunay criterion is not a mesh generation algorithm, it provides a criteria for which to connect vertices in a space. Given a set of vertices, a popular method to generate a mesh using the Delaunay criterion is the incremental Boywyer-Watson (Watson (1981)) algorithm, also developed by Hermeline (Hermeline (1982)). The algorithm starts with an initial simplex element (triangle in 2D, tetrahedra in 3D) that includes all the vertices. One vertex from the set is added at the time. If the circumsphere of an element contains the newly added vertex then the element is deleted. The deletion of elements using this criteria forms a contour cavity. Finally, the vertices of the contour cavity are connected with the inserted vertex to form new elements (Fig. 1.6).

In 2D, for a given number of points a mesh generated with connections handled with the Delaunay criteria can guarantee the maximization of the elements minimum angles which also minimizes the appearance of thin elements (sliver) which are often undesired. However this is not the case in 3D, as the criteria fails to spot every badly shaped thin tetrahedral element (sliver) and post-processing procedures are required to remove them. Such procedures include sliver exudation (Cheng et al. (2000)) that deletes sliver tetrahedral elements based on an extension of the Delaunay criterion and weight assignment to the points of elements that satisfy a ratio property or the use of variational methods that move the vertices, thus not preserving their original location (furtherly discussed in the following section). Based on the sliver exudation method of Cheng et al. (2000), another method for eliminating sliver tetrahedra is presented in Edelsbrunner et al. (2000). The method is based on perturbing the mesh vertices such that a new Delaunay tetrahedralization of the perturbed vertex set contains no slivers. More precisely, it is shown that a vertex can be moved within a sphere (perturbation ball) that does not contain forbidden regions. The forbidden regions are defined as volumes of tori containing the remaining vertices of the tetrahedra connected to the target vertex. If the target vertex is moved within the perturbation ball, any tetrahedra that are connected to it will not be sliver in the new tetrahedralization. The method assumes the mild perturbations of the mesh vertices do not depend on the direction the vertices are perturbed. Therefore, when boundaries are introduced and boundary vertices are either not allowed to move or move in restricted directions the method does not guarantee the elimination of sliver elements. A consequence of applying the method to boundary domains is that by perturbing the interior vertices for sliver elimination can lead to the possible appearance of sliver tetrahedra containing boundary vertices (Li (2001)).

Constrained Delaunay algorithms (CDT) (Paul Chew (1989), Shewchuk (2002a)) are applied to mesh the domain Ω of a geometry using the Delaunay criterion. Initially, a bounding box is created around a given discretised boundary geometry Γ (Fig.1.7a). An initial mesh is generated using the aforementioned incremental method to the set of points in Γ and the bounding box (Fig.1.7b). This initial mesh however does not guarantee extracting a mesh of Ω; the initial mesh may not contain facets (edges in 2D, faces in 3D) that are part of the boundary Γ. In 2D, iteratively swapping the element edges can recover the initial boundary (George et al. (1991)), while, in 3D, the same method may be insufficient to resolve this issue; additional facet operations along with insertion of new vertices may be required (Weatherill & Hassan (1994), George et al. (1991)) without though guaranteeing the recovery of the original boundary facets. In an attempt to conserve as much as possible the original boundary by avoiding the insertion of many additional vertices a Delaunay relaxation for the recovery of the boundary facets is also proposed (Shewchuk (2002b), Si & Gärtner (2005), Si & Shewchuk (2014)). Once all the facets of the discretized geometry are recovered, the elements located outside of Ω are deleted (Fig.1.7c).

The initial mesh generated of Ω contains all the boundary vertices. Such a mesh may include badly shaped elements. To improve the shape and size of the mesh's elements, additional vertices are inserted either on the boundary or the interior domain strategically though the process of Delaunay refinement and the connectivity is updated using the Delaunay criterion (Fig.1.7d). The insertion of vertices is based on two rules:

1. The diametrical circle of a constrained edge (edge that is not allowed to be flipped) is defined as the smallest circle that includes the edge. A constrained edge is said to be encroached if a vertex other than its endpoints is included on or inside its diametral circle (Ruppert (1993)) (Fig. 1.8a). Another criteria to determined if a constrained edge is encroached relies on checking if the edge is included a badly shaped element and its circumcenter lie on opposite sides of the edge (Chew (1989)) (Fig. 1.8b). Any encroached constrained edge is split into two edges by inserting a point in the middle of the edge .
2. Each badly shaped element (an element that has a circumradius-to-shortest edge ratio greater than some bound) is split by inserting a new vertex in new locations (e.g circumcenter of the element, centroid) to delete the element. If the new vertex encroachs upon any constrained edge then it is not added. Instead the constrained edges it would encroach upon are split.



(a)

(b)

Figure 1.8: (a) A constrained edge (highlighted) is encroached if a vertex is contained within its diametral circle (Ruppert (1993)). The constrained edge is split until no vertices are included in the diametral circle no constrained edge is encroached. (b) Alternatively an encroached constrained edge can be spotted if a badly shaped element (t) and its circumcenter (c) lie on opposite sides of it (Chew (1989)). All vertices in the encroached constrained edge's diametral circle are deleted and a point is inserted in the middle of it to form new elements.

The vertices added though this process are also known as Steiner points. The process of refinement aims at creating a mesh where badly shaped elements are avoided and to satisfy shape and size criteria. Common strategies for refinement include the insertion of Steiner points to the centroid of the elements (Weatherill & Hassan (1994)) or their circumhypersphere center (Chew (1989), Ruppert (1993)). Using the latter strategy triangles can be generated with a minimum bound on any angle in the mesh. Other methods insert new vertices at the end points of the segment connecting the center of two adjacent elements circumhypershere (Voronoi segment) (Rebay (1993)) or along the edges of the initial triangulation at a specified spacing ratio (George et al. (1991)).

In 2D, the convergence of CDT with the application of a refinement process is guaranteed and offers the creation of a mesh that satisfies user specified shape and size criteria. However, as mentioned, in 3D the Delaunay criteria does not suffice to guarantee the non existence of sliver elements. Moreover, boundary recovery may require complex procedures and is not always feasible. Overall, the problem of boundary recovery leads to an extensive increase in the complexity of a CDT algorithm.

1.3.4 Hybrid methods

The three aforementioned methods of mesh generation share their own advantages and drawbacks. Hybrid mesh generation methods combine seperate methods to benefit from the advantages of a method while avoiding the bottlenecks of another. Delaunay type methods have been combined with advancing front methods to place new vertices in a Delaunay mesh (Marcum & Weatherill (1995)). Vertices are inserted incrementally, but added from the boundary towards the interior. Each facet is examined to determine the ideal location for a new vertex on the interior of the existing Delaunay mesh. The vertex is then inserted and local reconnection is performed.

Conversely, the advance front method has been combined with the Delaunay criterion (Mavriplis (1995), Merriam (1991)); based on the advance front approach elements are incrementally created by connecting a facet with a vertex while the Delaunay criterion is used to adjust the placement of the vertex and adjust the connectivity of merging fronts. The use of such a method can help to overcome the problems that occur when fronts are merged using the classical approach.

Kd-trees are data structures used to subdivide point sets into blocks of unequal size each containing approximately equal number of points. The blocks are subdivided by alternatively splitting each dimension of the domain using a subdivision rule such as the number of points included in a block or by computing the median coordinate value of the splitting dimension. Kd-trees have been used in conjunction with Delaunay algorithms for mesh generation of large point data sets. The subdivision of the domain allows for a parallel implementation of Delaunay based algorithms to the points included in each block of the domain (Morozov & Peterka (2016), Guo et al. (2020)). To ensure that the final mesh conforms to the Delaunay criterion additional strategies are invoked. In Morozov & Peterka (2016) the size of the blocks is dynamically adapted to include neighbor vertices if a circumsphere of an element from the current Delaunay tessellation intersects with one of the neighbor cells. In Guo et al. (2020), once each block is meshed further mesh reconstruction procedures are followed to the overlapping areas based on constrained Delaunay and a graph cut algorithm. The overlapping areas are then merged with the meshed non overlapping regions to obtain the final mesh.

In Schroeder & Shephard (1990) an octree method is combined with the Delaunay criterion. The basic motivation is to build an octree procedure for octant geometries that can then be meshed using the Delaunay criterion. This hybrid approach keeps the spatial addressability, localized mesh control, geometric simplification features of the octree technique, while taking advantage of simple and optimal properties with the Delaunay triangulation. The major difficulty lies in maintaining the compatibility between octants because they are individually triangulated. Although hybrid methods achieve to combine the advantages of separate meshing algorithms, they can lead to the development of algorithms of higher complexity than their combined counterparts.

1.4 Mesh improvement

The most used methods to improve the quality of the mesh are smoothing and topological operations. Smoothing improves the quality of a mesh by repositioning its vertices without changing the connectivity either by moving them to the centroid of their adjoint vertices (Laplacian smoothing) or based on a optimization based algorithm (Fig. 1.9). Although Laplacian smoothing is effective for triangular meshes, it can fail to improve the quality of tetrahedral elements or to guarantee mesh validity. Alternatively, smoothing can be effective using numerical optimization on smooth objective functions defined on a local domain of low quality elements (local smoothing), such as maximizing the squared sum of the elements qualities that are connected to the vertex (Parthasarathy & Kodiyalam (1991)), local non smooth objective functions like maximizing the minimum angle of the connected elements (Freitag et al. (1995)) or on objective functions defined over the whole domain of the mesh (global smoothing). The latter case involves a series of mesh improvement methods also known as variational methods. Essentially, in the variational approach, the low quality mesh is used as a reference to generate a new mesh with improved quality under a coordinate transformation of its vertices which is determined by minimizing a mesh functional. For example, such mesh functionals include functionals based on the conditioning of the Jacobian matrix of the coordinate transformation (Knupp (1996), Knupp & Robidoux (2000)), functionals based on equidistribution and alignement conditions (Huang & Russell (2010), Huang (2001)), or the energy of harmonic mappings (Dvinsky (1991)).

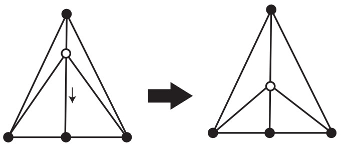


Figure 1.9: Example of smoothing operation. The vertex is repositioned to improve the quality of elements that are connected to it.

Local mesh improvement methods focus on improving the quality of a mesh by applying local smoothing coupled with geometrical operations on local mesh configurations of the mesh that include low quality elements. Topological operations change the topology of the mesh by removing a set of elements and replacing with another set occupying the same space. These include:

- **Flip operators**: The flip operator changes the connectivity locally by swapping the adjacent facets of elements. In 2D, the flip operator swaps the adjacent edge of the element. The 3D flip operator includes the 2-3 flip, 3-2 flip, 2-2 flip, and 4-4 flip (Freitag & Ollivier-Gooch (1997a)). The numbers denote the number of tetrahedra that are removed and created after flipping adjacent faces (Fig. 1.10).

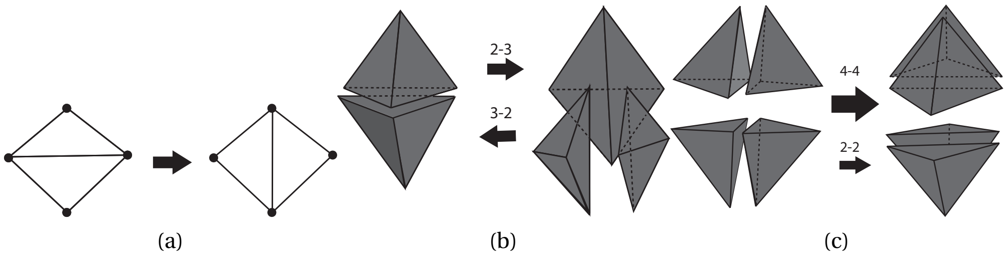


Figure 1.10: Examples of flip operator in 2D and 3D. In 2D, the flip operator changes the connectivity by swapping the adjacent edge of two elements (a). In 3D, there are several variants of the flip operator: the 2-3, 3-2 flip operators (b) and the 2-2, 4-4 flip operator (c) where the numbers denote the number of tetrahedra before and after applying the operation.

- **Edge removal**: Edge removal (de L'isle & George (1995)) starts by removing an edge from the mesh along with all the m tetrahedra that share the edge. Next, new connections are created by flipping edges that result in $2m-4$ tetrahedra. If a and b are the endpoints of the edge to be removed, the operation creates a new triangulation T (using a triangulation algorithm of Klincsek (1980)) for the domain R that is formed by the ring of vertices around the target edge ab. As a result the set I of tetrahedra that shared the edge before applying the operation are replaced by a set J of new tetrahedra of better quality (Fig. 1.11).

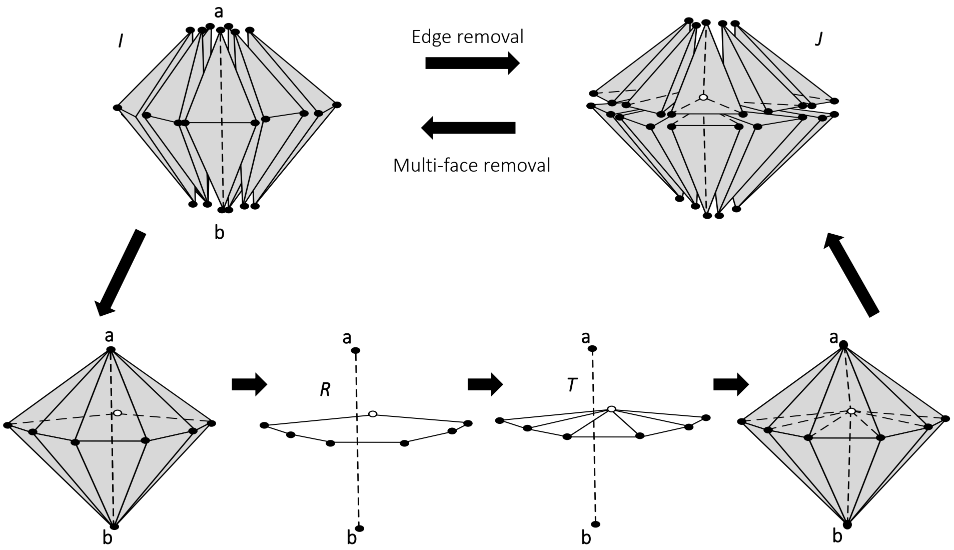


Figure 1.11: Edge removal and multi-face removal operations. Edge removal triangulates the domain R that contains the ring vertices around the edge ab which is an common edge for the tetrahedra in I. After the triangulation T the faces are connected with the vertices a and b to form the new set of tetrahedra J. Multi-face removal adjoins the sandwiched faces of the set of tetrahedra J, connects the a and b to form an edge, and connects the ring of vertices R with a and b to form the new set of tetrahedra I.

- **Multi-face removal**: Multi-face removal is the inverse operation of edge removal. It adjoins faces that are sandwiched between two endpoints of an edge to form new tetrahedra. If m faces are removed the $2m$ tetrahedra are replaced with $m+2$ (Fig. 1.11).
- **Vertex cavitation**: Vertex cavitation (Klingner & Shewchuk (2008)) builds a polyhedral cavity C around a vertex p and fills it with new elements by connecting the vertex with the polyhedral vertices (Fig. 1.12). p could be inserted or be part of an existing element. Unlike Delaunay refinement, the elements that are deleted to form C do not rely on a circum-hypersphere criterion but on an combinatorial optimization algorithm that maximizes the quality of the worst new element.

  The algorithm views the mesh as a graph with nodes that correspond to its elements and directed edges (u, w), if element u shares a facet (edge in 2D, face in 3D) with element w. u is considered the parent of w whereas w is the child of u. The elements that contain p are considered the root elements. Those elements are included in the cavity C. Next, starting from the facets of the root element (u) an adjacent element is visited (w). If by deleting the shared facet and connecting p with the vertices of the visited element w, the worst quality of the newly formed elements is higher than the quality of the root element then the adjacent element is included to C. Subsequently, all the children elements w of the parent adjacent element u that was included in C are visited. A children element u is added to C if by deleting all the facets that lead to it by following the directed edges from the root element and connecting the remaining vertices to p leads to the formation of new elements whose worst quality is better than the elements that were formed by its parent element w. It may possible that elements of u that are formed in the aforementioned fashion may be inverted, i.e topologically invalid, in which case u is not included in C. An additional stopping criterion for the formation of C is that the visit to new elements should not surpass a specific length following the directed graph from the root element.

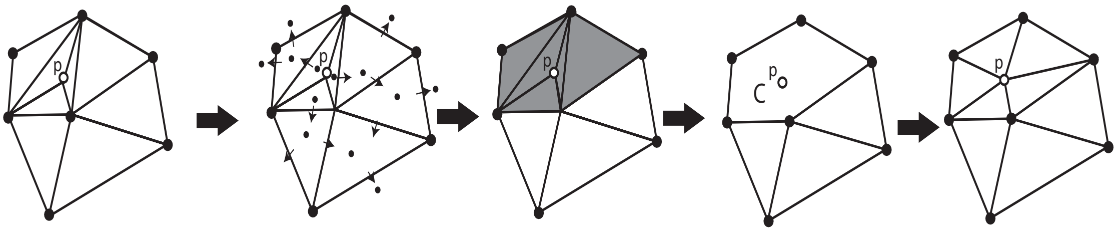


Figure 1.12: Example of vertex cavitation. The operation views the mesh as graph with nodes that correspond to the elements and directed edges that correspond to facet adjacent elements with a parent-child relation. Starting from the elements of p and following the directed edges, adjacent elements are visited to check whether they are included in the formation of a cavity C that includes p. The vertices of C are connected to p to form new elements.

Other operations change the topology of the mesh by adding new elements or deleting existing elements. Such operations include:

- **Vertex insertion**: Vertex insertion inserts a vertex at the barycenter either of a facet (edge insertion in 2D/3D, face insertion in 3D) or an element (face insertion in 2D,tetrahedral insertion in 3D) and connects the vertices of the element with it to form new elements (Fig .1.13).

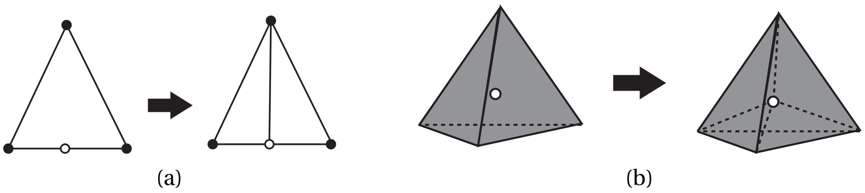


Figure 1.13: Vertex insertion operation. (a) Example of edge insertion. An vertex is inserted in the middle of an edge. The vertices of the element are connected to the inserted vertex to form new elements. (b) Example of tetrahedral insertion. A vertex is inserted in the barycenter of the tetrahedron and the vertices of the tetrahedron are connected with it.

- **Edge contraction**: Edge contraction removes an edge from the mesh by replacing the two endpoints with a single vertex (Fig. 1.14). As a result, the operation removes elements from the mesh that are unnecessarily too small. Additionally, the operation may improve the quality of a mesh as it removes elements that have a bad quality because they possess an edge that is too small.

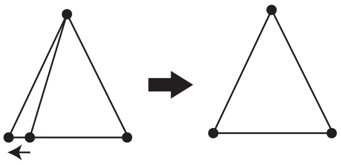


Figure 1.14: Example of edge contraction. An element is removed by collapsing its short edge.

In many cases, the application of solely one of the aforementioned operations is unable to improve the quality of a mesh. This leads to the application of an operation on top of an another one. For example, contraction is often followed by smoothing the vertices of the contracted edge. Similarly, after the application of vertex cavitation further topological and smoothing operations are applied to the retriangulated cavity. Compound operations (Tab. 1.1) are composed by the successive application of other base operations. As the application of base operations may lead to being stuck in a local optimum that is far from the global optimum, the application of compound operation leads the way to a better local optimum by "climbing" valleys of the objective function.

Category	Operation
Cavitation	**Edge Cavitation** : *Edge insertion* followed by *vertex cavitation* around the new vertex. **Face cavitation**: *Face insertion* followed by *vertex cavitation* around the new vertex. **Tetrahedron cavitation**: *Tetrahedron insertion* followed by *vertex cavitation* around the new vertex.
Contraction	**Face contraction**: *Edge insertion* followed by *edge contraction*. **Tetrahedron contraction**: *Face insertion* followed by *edge contraction* of the new edge. **Tetrahedron contraction**: *Edge insertion* on a edge followed by *edge insertion* on the opposite edge. Next, *edge contraction* is applied to the edge linking the two new vertices.
Smoothing	**Edge smoothing**: Performs the succesive *vertex smoothing* of the two vertices of an edge. **Tetrahedron smoothing**: Performs the succesive *vertex smoothing* of the four vertices of a tetrahedron.
Topological	**Multiface replacement**: *Multiface removal* followed by *edge removal*.

Table 1.1: Examples of compound operations composed by the successive application of other operations.

Local mesh improvement algorithms involve the application of schemes that apply a combination of mesh improvement operations using a hill climbing method; an operation is applied only if it improves the quality of the mesh. If the operation succeeds to improve the quality then another operation is applied for further improvement. As a result, the quality of the mesh after applying the improvement scheme cannot be worse than the original mesh. The mesh improvement scheme stops when the quality of the mesh cannot be further improved or if the quality gain to computational time ratio is small.

The *Stellar* improvement scheme (Klingner & Shewchuk (2008)) is an example of a local mesh improvement scheme for tetrahedral meshes. The scheme relies on improvement passes during which a list of operations solely of a specific category (cavitation, contraction, smoothing or topological) is applied to elements that either are below a quality threshold or have edges that are too large or too small. Initially, the improvement scheme builds a list of elements E that are below a quality threshold (*goalQuality*) (Alg. 1, Line 2). Subsequently, the *SmoothingPass* procedure (Alg. 1, Line 3) is called, during which the operation of *vertex smoothing* is performed to all the vertices of the E once. Then, the procedure *TopologicaPass* is called (Alg. 1, Line 4) which applies topological operations, for example *edge removal*, on every element in the E. Next, the *ContractionPass* procedure(*Alg. 1, Line 5*) is called, which applies *edge contraction*.

The scheme proceeds to regulate the edge length of the elements according to *shortGoalLength* and *LongGoalLength* by calling the *SizingControl* procedure. At the beginning of this procedure, edges that are too short are contracted by calling the *ContractionPass* procedure (Alg. 2, Line 4). Then, the edges that are too long are split using the *CavitationPass* procedure (Alg. 2, Line 5). If the shortest and longest edges didn't change after the previous passes, a *SmoothingPass* and a *TopologicalPass* is applied to improve the quality of the elements in E (Alg. 2, Lines 6-9). After improving the quality of the elements, the edge lengths are regulated again. The sizing loop is repeated multiple times until a desirable short and long edge length is met. Once the *SizingControl* procedure is finished, the quality of the elements in E by applying a loop of improvement passes (Alg. 1, Lines 7-23) consisting of a *SmoothPass*, a *TopologicalPass*, a *ContractionPass*, and a *CavitationPass*. The loop is repeated until a certain goal quality is met or a number of improvement passes is exceeded.

Algorithm 1: Stellar Improvement Scheme

```
StellarImprovement (maxNumOfPasses, goalQuality)
    build list of elements E with a quality below goalQuality
    Call procedure SmoothingPass
    Call procedure TopologicalPass
    Call procedure ContractionPass
    Call procedure SizingControl
    while $numOfPasses < maxNumOfPasses$ do
        if $minQuality \geq goalQuality$ then
            return
        end
        else
            Call procedure SmoothingPass
            if minQuality has not improved and meanQuality has not improved then
                Call procedure TopologicalPass
            end
            if minQuality has not improved and meanQuality has not improved then
                Call procedure ContractionPass
            end
            if minQuality has not improved and meanQuality has not improved then
                Call procedure CavitationPass
            end
        end
    end
```

Algorithm 2: Sizing control

```
SizingControl (shortGoalLength, longGoalLength, maxIterations)
    while ($minEdgeLength < shortGoalLength$ or $maxEdgeLength > longGoalLength$) and ( $numIterations < maxIterations$) do
        Grab all too short and too long edges in the mesh
        Call procedure ContractionPass
        Call procedure CavitationPass
        if minEdgeLength and maxEdgeLength didn't change then
            call procedure SmoothingPass on every bad element in E
            call procedure TopologicalPass on every bad element in E
        end
    end
```

The *Stellar* improvement scheme is only an example among various schemes. In principle, the selection of the series of operations to be applied make local mesh improvement a combinatorial optimization problem; The final mesh that emerges after applying an improvement scheme is dependent on the order of the operations. For example, if the *TopologicalPass* and *CavitationPass* had a head priority over the *SmoothingPass*, a different mesh would emerge after the application of the scheme. It is unclear how a particular order of a selection influences the optimization path and how it differs from another selection

Other heuristic parameters for mesh improvement include:

- Parameters associated with a mesh improvement operation the selection of which can influence the result of the final mesh. For example, in vertex cavitation the bigger the cavity is artificially expanded, there are more changes that subsequent topological operations to the remeshed cavity are able to improve the quality of the mesh. However, the bigger a cavity is expanded the more computational time is required for the operation to be applied.
- When a mesh improvement operation is applied to an element, it is possible that it affects its adjacent elements. For the mesh improvement scheme to be successful it is essential to improve the affected elements as well. The depth of an improvement loop determines how many recursive calls are initiated to improve the quality of those affected elements when an unsuccessful operation is applied. When the depth is increased the computational time can increase exponentially. It is unclear which level of depth we must reach with the hope of improving a local mesh configuration.
- The choice of successive operations following the unsuccessful one in the case of compound operations is also unclear. For example, the application of smoothing operations on the affected elements after the application of a topological operation, beats the purpose of applying it in the first place, since topological operations affect the connectivity and not the position of the vertices.
- The result of the final mesh can differ by applying a list of operations to each element under a quality threshold or applying the same operation to all of them.
- The priority in which elements below a quality threshold are improved affects the overall outcome as well. For example, in the case of the smoothing operation, sorting element from worst to best quality can restrict the range which the vertices of the elements can be moved.

Overall the aforementioned parameters render the development of local mesh improvement schemes that is optimal in terms of quality and computational time a complex task. Therefore, although the local approach of improvement schemes are favorable for the improvement of Lagrangian meshes, where the smoothing of the field values needs to be avoided as much as possible, they may lead to the development of complicated algorithms that account for a significant computational time during the course of a simulation and are not always robust.

1.5 Artificial Neural Networks

Machine learning refers to an artificial intelligence (AI) field of study that enables systems to automatically learn and improve from experience without or with little explicit human interference. It focuses on the development of algorithms that acquire data and build models

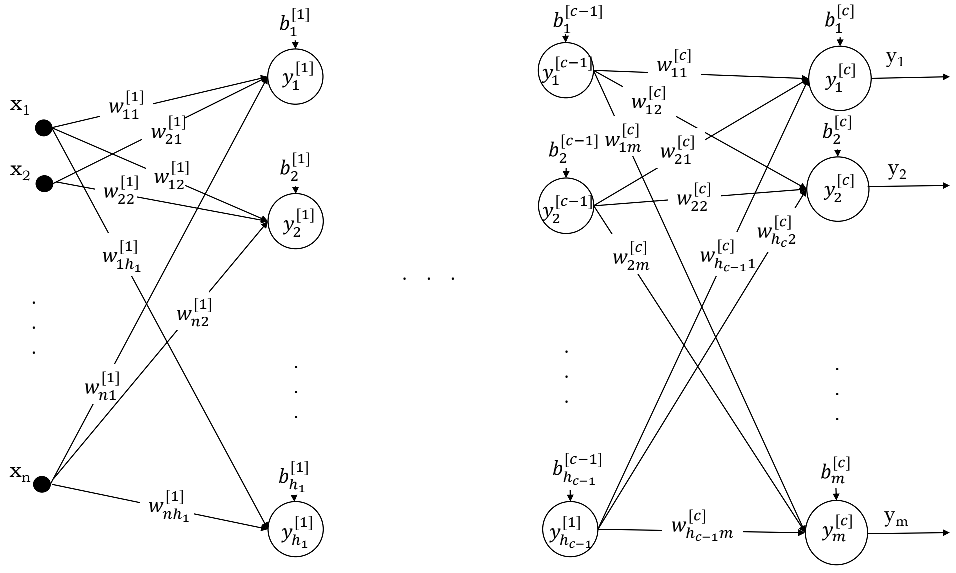


Figure 1.15: Neural network with feed forward architecture. Neurons are grouped into a sequence of c layers $L^{[1]},...,L^{[c]}$. The hidden layers $L^{[1]},...,L^{[c-1]}$ contain $h_1,...,h_{c-1}$ hidden neurons, and the output layer $L^{[c]}$ is composed of m output neurons. Each layer is associated with a set of free parameters $a = (a^{[1]},..,a^{[c]})$. For every layer l the free parameters are defined as the pair $a^{[l]} = (w^{[l]}, b^{[l]})$, where $w^{[l]} = (w_{1,1}^{[l]}, w_{1,2}^{[l]},.., w_{1,h_l}^{[l]},..., w_{h_{l-1},1}^{[l]},..., w_{h_{l-1},h_l}^{[l]})$ are the weight synapses, $b^{[l]} = (b_1^{[l]},.., b_{h_l}^{[l]})$ are the biases, and h_l is the number of neurons of the $L^{[l]}$ layer. The output of each neuron is an output of a non linear function $y_j^{[l]}$, where $l \in \{1,2,...,c\}$, $j \in \{1,2,...,h_l\}$.

in order to make better decisions according to prior observation or data records.

According to the learning method that is adopted, machine learning algorithms are usually categorized as being either supervised or unsupervised. In supervised learning, a model at hand is trained using a certain data set along with its respective labels. Thus, once a model is trained on known data, it can be further used with another set of data to infer their labels. In unsupervised learning, however, prior labels are inaccessible or accessible but unimportant for the application being addressed. This latter, thus, consists in studying how systems can infer functions to define hidden structures from unlabeled data. Semi-supervised learning is another direction whose aim is to exploit a small-sized label data and a large-sized unlabeled data

A popular set of supervised and unsupervised learning algorithms are artificial neural networks. There are many different types of neural networks. In supervised learning, the feed forward Neural Network, also known as multilayer perceptron, is an important one, and most of the literature in the field is commonly referred to this as an artificial neural network (NN). A NN is characterized by a set of neuron models that are interconnected forming an architecture.

Hence, neurons in a feed-forward NN are grouped into a sequence of c layers $L^{[1]},...,L^{[c]}$, so that neurons in any layer are connected only to neurons in the next layer. The input layer $L^{[0]}$ consists of n external inputs and is not counted as a layer of neurons; the hidden layers $L^{[1]},...,L^{[c-1]}$ contain $h_1,...,h_{c-1}$ hidden neurons, respectively, and the output layer $L^{[c]}$ is composed of m output neurons (Fig. 1.15). Communication proceeds layer by layer from the input layer via the hidden layers up to the output layer. The states of the output neurons represent the result of the computation.

A feed forward Neural Network architecture defines a non linear function composition:

$$y(X;a) = y^{[c]}((y^{[c-1]}(..y^{[2]}(y^{[1]};a^{[1]});a^{[2]});a^{[c-1]});a^{[c]}) \tag{1.1}$$

where $X = (x_1,x_2,...,x_n)$ is an input signal and $a = (a^{[1]},..,a^{[c]})$ is a set of free parameters associated with each layer. For every layer $L^{[l]}$ the free parameters are defined as the pair $a^{[l]} = (w^{[l]},b^{[l]})$, where $w^{[l]} = (w^{[l]}_{1,1}, w^{[l]}_{1,2},.., w^{[l]}_{1,h_l},..., w^{[l]}_{h_{l-1},1},..., w^{[l]}_{h_{l-1},h_l})$ are the weight synapses, $b^{[l]} = (b^{[l]}_1,..,b^{[l]}_{h_l})$ are the biases and h_l is the number of neurons at the $L^{[l]}$ layer. The output of each neuron is an output of a non linear function $y^{[l]}_j$, where $l \in \{1,2,...,c\}$, $j \in \{1,2,...,h_l\}$.

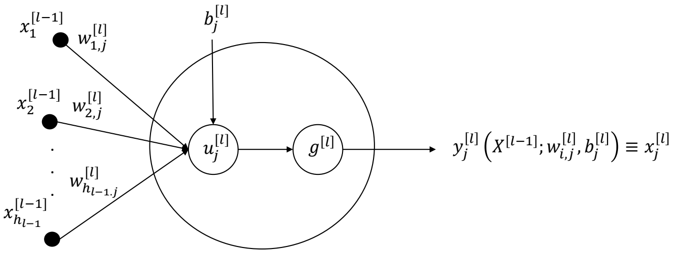


Figure 1.16: Perceptron neuron model. The input signals $X^{[l-1]} = (x^{[l-1]}_1,..,x^{[l-1]}_{h_{l-1}})$ from $L^{[l-1]}$ layer are transformed using the activation function to the signal $u^{[l]}_j = b^{[l]}_j + \sum_{i=1}^{h_l} w^{[l]}_{i,j} x^{[l-1]}_i$. The outcome of $u^{[l]}_j$ is then given as input to the activation function $g^{[l]}$ that defines the output signal $y^{[l]}_j$. The output signal $y^{[l]}_j$ becomes an input signal $x^{[l]}_j$ for neurons of the $L^{[l+1]}$ layer.

The input of the j^{th} neuron in the $L^{[l]}$ layer is obtained by forming the a linear combination of the $h_{[l-1]}$ input signals $X^{[l-1]} = (x^{[l-1]}_1,..,x^{[l-1]}_{h_{l-1}})$ from the $L^{[l-1]}$ layer:

$$u_j^{[l]} = b_j^{[l]} + \sum_{i=1}^{h_l} w_{i,j}^{[l]} x_i^{[l-1]} \tag{1.2}$$

, u is also known as the combination function. The output of the neuron is the non linear function:

$$y_j^{[l]}(X^{[l-1]}; w_{i,j}^{[l]}, b_j^{[l]}) \equiv x_j^{[l]} = g^{[l]}(u_j^{[l]}) = g^{[l]}(b_j^{[l} + \sum_{i=1}^{h_l} w_{ij}^{[l]} x_i^{[l-1]}) \tag{1.3}$$

, where g is known as the activation function of the neuron. The output signal of the neuron is subsequently transferred as an input signal $x_j^{[l]}$ for neurons of the next layer $l+1$ (Fig. 1.16). Three of the most used activation functions (Fig. 1.17) include:

- The logistic function:

$$g(u) = \frac{1}{(1+e^{-u})} \tag{1.4}$$

- The hyperbolic tangent function

$$g(u) = \tanh(u) = \frac{(e^u - e^{-u})}{(e^u + e^u)} \tag{1.5}$$

- The Rectified Linear Unit (ReLU)

$$g(u) = \begin{cases} u, & \text{for } u \geq 0 \\ 0, & \text{for } u < 0 \end{cases} \tag{1.6}$$

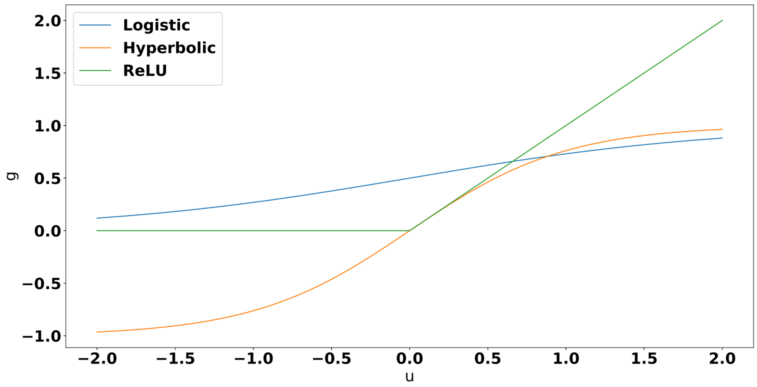


Figure 1.17: Graphs of the logistic, hyperbolic, and ReLU activation functions.

Activation functions allow the NNs to create complex mappings between the network's inputs and outputs, which are essential for learning and modeling complex data, such as images, video, audio, and data sets which are non-linear or have high dimensionality. Almost any process imaginable can be represented as a functional computation in a NN, provided that the activation function is non-linear.

The choice of activation function plays an essential role for the training process of an NN (see next section) to predict values where gradients are updated through an iterative process. The logistic function is smooth and derivable at every point, which is a desirable property for any activation function. The output of the logistic is normalized in the range 0 to 1. The logistic function is mostly used before the output layer for classification . The hyperbolic tangent function has a similar behavior with logistic function. The output of the logistic is normalized in the range -1 to 1. Unlike the logistic function the hyperbolic tangent function is zero centered making it easier to model inputs that have strongly negative, neutral, and strongly positive values. Both the logistic and hyperbolic functions can cause a vanishing gradient problem that renders the gradient update procedure of the training process inefficient (especially for NNs with large number of layers), which in turn leads to inaccurate predictions of the NN. To overcome the problem of vanishing gradient, the ReLU function is preferred for NNs with a large number of layers. Moreover, compared to the the aforementioned activation functions that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero. However, when inputs approach zero, or are negative, the gradient of the function becomes zero, the network causing issues to the training procedure.

1.5.1 Training process

Given a dataset $D = \{(X^{(k)}, Y^{(k)})\}_{k=0}^{N}$, where $X^{(k)} = (x_1^{(k)}, .., x_n^{(k)})$ are the per sample input signals and $Y^{(k)} = (y_1^{(k)}, .., y_m^{(k)})$ are the desirable outputs, a NN undergoes a training process during which its parameters a are adjusted such that the non linear function $y(X; A)$ best fits the set of desirable outputs. For a set of free NN parameters $a = (w, b)$ the approximation $\hat{Y}$ of the desirable output Y is measured using a loss function $\mathscr{L}(\hat{Y}, Y; A)$, also known as objective functional.

The training process is equivalent to finding a set of free parameters A through an optimization process such that to minimize the loss function:

$$\mathscr{L}(\hat{Y}, Y; a) = \mathscr{L}(y(X; a), Y) = \mathscr{L}(y(X; w, b), Y) = \sum_{k=1}^{N} \ell(y(X^{(k)}; w, b), Y^{(k)}) \tag{1.7}$$

, where $\ell(y(X^{(k)}; w, b) = \ell_k$ is the per sample loss function.

Gradient descent is the most popular optimization strategy used for the training of NNs. To minimize a functional $\mathscr{L} : \mathbb{R}^D \to \mathbb{R}$, gradient descent uses a local linear information to iteratively move toward a local minimum. Through the iterative process of free parameter update:

$$a_{t+1} = a_t - \eta \nabla \mathscr{L}(a_t) \tag{1.8}$$

the local minimum of $\mathscr{L}$ is found (Fig. 1.18). The choice of the initial value of w_0 and η, also known as learning rate, define how the process of finding the local minimum behaves. The initial value w_0 affects the number of iterations to reach to a minimal of the objective functional. If the value of η is too small the convergence to the minimal of the objective function can be slow whereas if it's too big gradient descent can fail to converge to the minimal or even diverge.

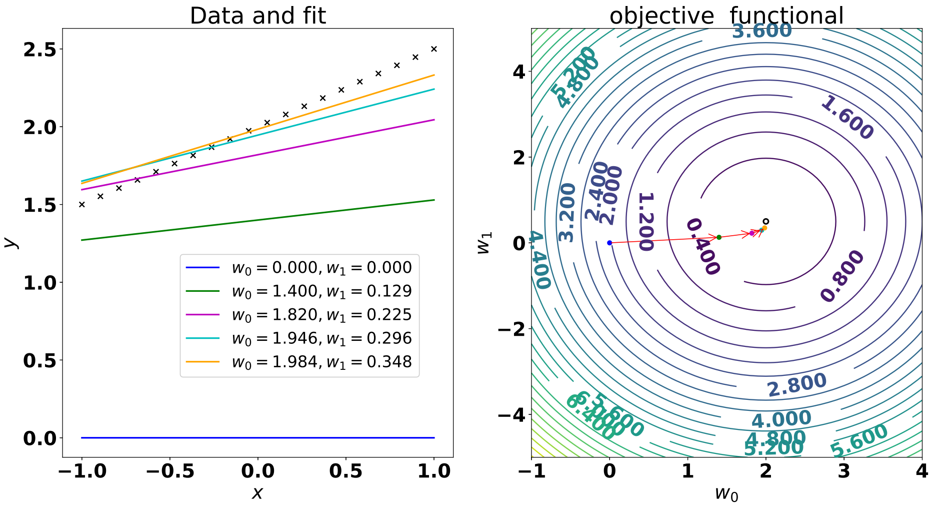


Figure 1.18: Example of gradient descent for fitting a giving dataset $(X^{(i)}, Y^{(i)})$, where $i = \{1, 2, ..m\}$, to a straight line $h_a(x) = w_0 + w_1 x$. The free parameters $a = (w_0, w_1)$ are updated through the iterative process $w_{j,t+1} = w_{j,t} - \eta(\partial \mathscr{L}(w_t)/\partial w_j)$, where $j = \{0, 1\}$, using the objective functional $\mathscr{L}(w_t) = (1/2m)\sum_{i=0}^{m}(h_a(X^{(i)} - Y^{(i)})^2$. At each iteration the free parameters a get closer to the minima of the objective functional and the line fits better the dataset.

To apply gradient descent to the loss function of eq.1.7 the expression of the gradient of the per sample loss function $\ell_k = \ell(y(X^{(k)}; w, b), Y^{(k)})$ with respect to the parameters $a = (w, b)$ must be calculated, e.g.:

$$\frac{\partial \ell_k}{\partial w_{i,j}^{[l]}} \quad \text{and} \quad \frac{\partial \ell_k}{\partial b_i^{[l]}} \tag{1.9}$$

for $l = 1, 2, ..., c$.

To compute the derivatives of eq.1.9 and apply gradient descent the back propagation algorithm is applied.

For a training dataset $D = \{(X^{(k)}, Y^{(k)})\}_{k=0}^{N}$, for every layer $L^{[l]}$, where $l = 1, 2, ...c$ the derivative of the per sample loss function $\ell_k = \ell(y(X^{(k)}; w, b), Y^{(k)})$, $\partial \ell_k / \partial w_{i,j}^{[l]}$ and $\partial \ell_k / \partial b_j^{[l]}$, can be stored using a Jacobian matrix:

$$\left[\frac{\partial \ell_k}{\partial b^{[l]}}\right] = \begin{bmatrix} \frac{\partial \ell_1}{\partial b_1} & \frac{\partial \ell_1}{\partial b_2} & \cdots & \frac{\partial \ell_1}{\partial b_{h_l}} \\ \frac{\partial \ell_2}{\partial b_1} & \frac{\partial \ell_2}{\partial b_2} & \cdots & \frac{\partial \ell_1}{\partial b_{h_l}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \ell_N}{\partial b_1} & \frac{\partial \ell_N}{\partial b_2} & \cdots & \frac{\partial \ell_N}{\partial b_{h_l}} \end{bmatrix}$$

and using the compact notation :

$$\left[\left[\frac{\partial \ell_k}{\partial w^{[l]}}\right]\right] = \begin{bmatrix} \frac{\partial \ell_1}{\partial w_{1,1}} & \frac{\partial \ell_1}{\partial w_{1,2}} & \cdots & \frac{\partial \ell_1}{\partial w_{1,h_l}} \\ \frac{\partial \ell_2}{\partial w_{2,1}} & \frac{\partial \ell_2}{\partial w_{2,2}} & \cdots & \frac{\partial \ell_2}{\partial w_{2,h_l}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \ell_N}{\partial w_{h_{l-1},1}} & \frac{\partial l_n}{\partial w_{h_{l-1},2}} & \cdots & \frac{\partial l_n}{\partial w_{h_{l-1},h_l}} \end{bmatrix}$$

The values of the partial derivatives can then be used to update the free parameters $a = (w, b)$ of the NN though with gradient descent as in eq.1.8:

$$w^{[l]} \leftarrow w^{[l]} - \eta \sum_n \left[\left[\frac{\partial \ell_n}{\partial w^{[l]}}\right]\right] \quad \text{and} \quad b^{[l]} \leftarrow b^{[l]} - \eta \sum_n \left[\frac{\partial \ell_n}{\partial b^{[l]}}\right] \tag{1.10}$$

The training procedure is carried out by the back propagation algorithm. For a NN consisting of $L^{[c]}$ layers the back propagation algorithm (see **Appendix A.1**) for a input training signal $X = (x_1, x_2, .., x_n)$ can be summarized as followed:

(i) **Forward pass** : During this phase the signal is propagated through the network, $\forall l = 1, 2, ..., c$:

$$u^{[l]} = w^{[l]} X^{[l-1]} + b^{[l]} \tag{1.11}$$

and

$$X^{[l]} = g^{[l]}(u^{[l]}) \tag{1.12}$$

is calculated.

(ii) **Backward pass**: Starting from the last layer $L^{[c]}$ we compute the Jacobian matrix:

$$\left[\frac{\partial \ell}{\partial X^{[c]}}\right] = \nabla \ell(X^{[c]}) \tag{1.13}$$

where $X^{[c]} = (x_1^{[c]}, x_2^{[c]}, ..., x_{h_c}^{[c]})$ are the output signals. Then for the layers $L^{[c-1]}, ..., L^{[1]}$:

$$\left[\frac{\partial \ell}{\partial X^{[l]}}\right] = (w^{[l+1]})^T \left[\frac{\partial \ell}{\partial u^{[l+1]}}\right] \tag{1.14}$$

$$\left[\frac{\partial \ell}{\partial u^{[l]}}\right] = \left[\frac{\partial \ell}{\partial X^{[l]}}\right] \odot \dot{g}^{[l]}(u^{(l)}) \tag{1.15}$$

from eq. 1.13, 1.14 and 1.15 the partial derivatives of the loss function with respect to the free parameters of the NN can be acquired:

$$\left[\left[\frac{\partial \ell}{\partial w^{[l]}}\right]\right] = \left[\frac{\partial \ell}{\partial u^{[l]}}\right] (X^{[l-1]})^T \tag{1.16}$$

$$\left[\frac{\partial \ell}{\partial b^{[l]}}\right] = \left[\frac{\partial \ell}{\partial u^{[l]}}\right] \tag{1.17}$$

(iii) **Gradient step**: Using 1.16 and 1.17 the weights and biases of the NN are updated using gradient step as in eq.1.10.

(iv) **Iteration**: The steps (i)-(iii) are repeated until specified termination criteria are met. This criteria are either based on an error threshold or the number of iterations.

1.5.2 Hyperparameters

Hyperparameters are the set of parameters that influence the behavior of gradient descent for the minimalization of the loss function $\mathscr{L}$. A set of such parameters involves the learning rate η, the initial values of the weights w_0, the choice of activation function, the number of hidden neurons and the number of hidden layers. Another parameter that dictates the training behavior of the NNs is the size of the training data that partakes in the free parameter $a = (w, b)$ update. For the training of the NNs, three main variants of gradient descent are generally applied based on the size of training data. Based on the amount of training data a trade off is made between the accuracy of the parameter update and the computational time of the update.

The standard gradient descent algorithm, also known as batch gradient descent, computes the gradient of the loss function with respect to the parameters $a = (w, b)$ for the entire training dataset:

$$a_{t+1} = a_t + \eta \nabla_{a_t} \mathscr{L}(a_t) = a_t + \eta \sum_{k=1}^{N} \nabla \ell_k(y(X^{(k)}; a_t), Y^{(k)}) \tag{1.18}$$

Since the gradients of the whole dataset need to be computed to perform just one update, batch gradient descent can account for big computation times and can be hard to manage for resources that don't meet the memory requirements of the dataset. Moreover, the gradient is computed incrementally so by the time $\nabla \ell_k$ is calculated, $\nabla \ell_{k-1}, \nabla \ell_{k-2}, \dots \nabla \ell_1$ have already been computed; therefore, better estimate $\hat{a}$ of a_t could be achieved. Despite all of the drawbacks, batch gradient descent has been proven to converge to a local or global minima for problems of convex optimization (Fig 1.19).

In contrast to bath gradient descent, stochastic gradient descent (SGD) updates the parameters for each training example $X^{(k)}$ and $Y^{(k)}$, where $k = 1, 2, \dots N$:

$$a_{t+1} = a_t + \eta \nabla \ell_k(y(X^{(k)}; a_t), Y^{(k)}) \tag{1.19}$$

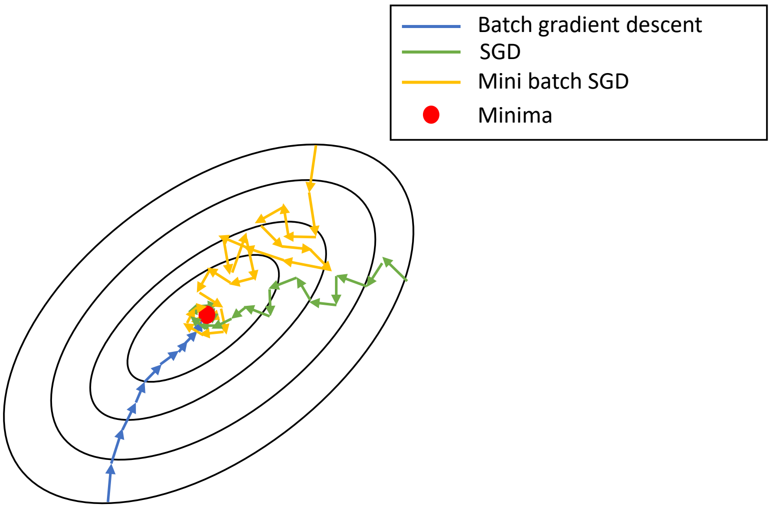


Figure 1.19: Optimization path using batch gradient descent, SGD and mini batch SGD for a convex loss function. Batch gradient descent, although computationaly expensive, is proven to converge to the minimal of convex loss funtions. SGD is faster, however, due to its nature, the optimization path may oscillate and overshoot near the minima. Mini batch SGD can be computationally efficient and can reduce the oscillations of SGD.

SGD is much faster than batch gradient descent; batch gradient descent may suffer from redundant computations if the training dataset contains similar examples. SGD avoids the redundant computations by performing one parameter update at the time. A consequence of the stochastic step is that the parameter updates may lead to an oscillated trajectory to the minima of the objective function due to high variance between steps (Fig 1.19). However, this fluctuations enable SGD to a potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, when the learning rate is decreased, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

Mini batch SGD is a mixture of SGD and batch gradient descent. At each step, mini batch SGD performs parameter update for a mini batch of B training example batches:

$$a_{t+1} = a_t + \eta \nabla \sum_{b=1}^{B} \ell_{k(t,b)}((y(X^{k(t,b)}; a_t), Y^{k(t,b)}) \tag{1.20}$$

The order to visit the batch samples $k_{(t,b)}$ can either be sequential or with uniform sampling without replacement. The aforementioned parameter update reduces its variance leading to a more stable convergence (Fig 1.19). Additionally, the separation of the training dataset into mini batches allows for it to fit in processing memory taking advantage of computational resources and also for the parallelization of the gradient computation for each batch.

Another hyperparameter involves the optimization method used to reach the local minima while using mini batch SGD to reduce the oscillating behavior of the optimization path and reach faster to the minimal of the loss function. A first improvement in the optimization of the mini batch SGD involves adding a momentum term that accelerates in the direction of finding a minima of the loss function (Fig. 1.20). This is done by adding a term m during the update process which adds inertia to the step direction. If $g_t = \sum_{b=1}^{B} \ell_{k(t,b)}((y(X^{k(t,b)}; a_t), Y^{k(t,b)})$ and $\gamma > 0$ then the parameters are updated as:

$$m_t = \gamma m_{t-1} + \eta g_t \tag{1.21}$$

$$w_{t+1} = w_t - m_t \tag{1.22}$$

Another class of methods exploits the statistics over the previous steps to compensate for the anisotropy of the loss function contours. The Adam optimization method uses the first and second moments of the gradient m_t an v_t respectively (Fig. 1.20). The first moment involves the exponentially decaying average of the previous gradient (similar to momentum) while the second moment involves the decaying average of the previous squared gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{1.23}$$

$$v_t = \beta_2 v_{t-1} + \beta_2 g_t^2 \tag{1.24}$$

Because m_t and v_t are initialized as zero value vectors, during the initial steps they are biased towards zero. To counteract these biases the bias corrected first and second moment estimates $\hat{m}_t$ and $\hat{v}_t$ respectively are computed:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{1.25}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{1.26}$$

Using $\hat{m}_t$ and $\hat{v}_t$ yields the update rule :

$$a_{t+1} = a_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{1.27}$$

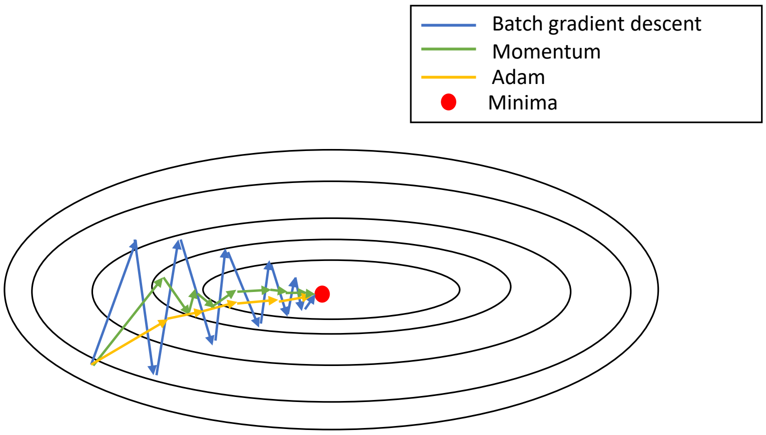
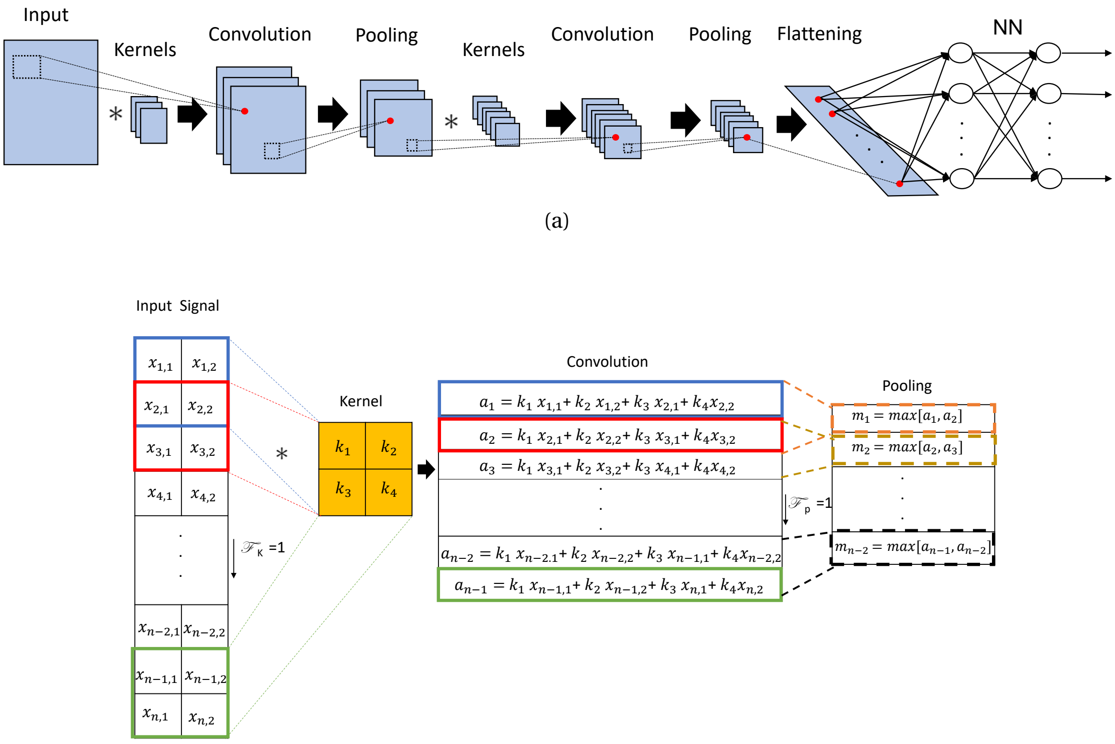


Figure 1.20: Optimization path using mini batch SGD with different optimization strategies. Momentum and Adam add an additional computational cost at each free parameter update step but reduce the oscillating behavior of mini batch SGD making it possible to reach faster the minima.

The term generalization in supervised machine learning algorithms refers to their ability to infer new information of unknown data; after being trained on a given dataset, an algorithm can successfully make accurate predictions on new data. However, a model can make inaccurate predictions if it has not been trained enough on the training data (underfitting). The inverse is also possible; a model can predict inaccurately if it has been trained too much on the training data. The latter case is also known as overfitting. To this end, regularization techniques aim at reducing the generalization error but not the training error. Such a technique involves regulating the values of the weights in a neural network. It has been observed that NNs with smaller weights tend to generalize better and overfit less. Thus, to avoid overfitting a weight term $0 < \lambda < 1$ can be added during the update step of the weights during gradient descent. λ is referred to as weight decay. According to eq.1.27 the introduction of regularization term yields the following update step for the weights:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t - \eta \lambda w_t \tag{1.28}$$

1.5.3 Convolution


Input
Kernels
Convolution
Pooling
Kernels
Convolution
Pooling
Flattening
NN
*
*
Input Signal
$x_{1,1}$ $x_{1,2}$
$x_{2,1}$ $x_{2,2}$
$x_{3,1}$ $x_{3,2}$
$x_{4,1}$ $x_{4,2}$
$x_{n-2,1}$ $x_{n-2,2}$
$x_{n-1,1}$ $x_{n-1,2}$
$x_{n,1}$ $x_{n,2}$
$\mathscr{F}_K = 1$
Kernel
k_1 k_2
k_3 k_4
Convolution
$a_1 = k_1 x_{1,1} + k_2 x_{1,2} + k_3 x_{2,1} + k_4 x_{2,2}$
$a_2 = k_1 x_{2,1} + k_2 x_{2,2} + k_3 x_{3,1} + k_4 x_{3,2}$
$a_3 = k_1 x_{3,1} + k_2 x_{3,2} + k_3 x_{4,1} + k_4 x_{4,2}$
$a_{n-2} = k_1 x_{n-2.1} + k_2 x_{n-2,2} + k_3 x_{n-1,1} + k_4 x_{n-2,2}$
$a_{n-1} = k_1 x_{n-1,1} + k_2 x_{n-1,2} + k_3 x_{n,1} + k_4 x_{n,2}$
$\mathscr{F}_p = 1$
Pooling
$m_1 = max[a_1, a_2]$
$m_2 = max[a_2, a_3]$
$m_{n-2} = max[a_{n-1}, a_{n-2}]$

(a)

(b)

Figure 1.21: (a) Example of a convolutional NN. Multiple kernels are used to apply convolution to the input signal resulting in multiple convoluted feature maps. Each of convoluted feature map undergoes a pooling process. The process of convolution and pooling can repeat before flattening the result and connecting it with an NN. (b) Example of convolution and pooling operation to an input signal of signal $n \times 2$. Using a kernel K of size 2×2 with a stride $\mathscr{F}_K = 1$ the convoluted signal has a size $(n-1) \times 2$. Applying the max pooling layer with a stride $\mathscr{F}_p = 1$ results in a signal $(n-2) \times 2$.

NNs often feature a convolution layer that processes the multidimensional input signals by applying the same linear transformation locally, enabling weight sharing. Convolution layers have been proven to improve the accuracy of NNs with respect to image, text and signal processing input. Weight sharing of the signal input allows for input feature extraction by modeling automatically local correlations. Convolution layers are often combined with pooling layers that downscale the convoluted result and preserve the signal structure. A convolutional NN (CNN) is a NN composed of a convolution layer that processes the input features through multiple stages of convolution and pooling layers before being connected to a feed forward architecture (Fig. 1.21a). CNNs are designed to automatically and adaptively learn spatial hierarchies of features and are commonly applied to input signals with a grid like structure (e.g pixels of an image).

In the context of a(CNN), a convolution is a linear operation that involves the multiplication of a set of weights with the input signal, much like a traditional NN. The multiplication is performed between an array of input signal and a array of weights, called a kernel. The kernel has a smaller dimension than the input signal and the type of multiplication applied between a filter-sized patch of the input and the kernel is element wise. If the kernel is designed to detect a specific type of feature in the input signal, then the application of that kernel systematically across the entire input signal allows the filter an opportunity to discover that feature anywhere in the signal. For example, with the use of convolution, local patterns in an image can be discovered even if the image is subjected to distortions (e.g rotation, scaling). In 2D convolution, if the /input signal is of size $S = C \times H$ and the convolution kernel is a signal of size $s = c \times h$ then the output will be of size $(C - c + 1/\mathscr{F}_K) \times (H - h + 1/\mathscr{F}_K)$, where $\mathscr{F}_K$ is the step size when moving the kernel across the signal (stride) (Fig. 1.21b). Given an input $X \in \mathbb{R}^S$ and convolution kernel $K \in \mathbb{R}^s$, the convolution matrix $\mathscr{A}$ is defined as:

$$\mathscr{A}(i, j) = \sum_{m=0}^{C-1} \sum_{n=0}^{H-1} X(m, n) K(i - m, j - n), \; 0 < i < C + c - 1 \,, 0 < j < H + h - 1 \tag{1.29}$$

The 2D convolution matrix $\mathscr{A}$ of size $C_K \times H_K$, where $C_K = (C - c + 1/\mathscr{F}_K)$ and $H_K = (H - h + 1/\mathscr{F}_K)$, undergoes a pooling process. A pooling layer provides a typical downsampling operation which reduces the dimensionality of the feature maps in order to introduce a translation invariance to small shifts and distortions, and decrease the number of learnable parameters. Given a pooling area size $h_p \times w_p$ and a stride $\mathscr{F}_p$ there are two main types of pooling producing a signal of size $(C_K - c_p + 1/\mathscr{F}_p) \times (H_K - h_p + 1/\mathscr{F}_p)$. Max-pooling is defined as:

$$M(i, j) = \max_{0 \le m < c_p} \max_{0 \le n < h_p} \mathscr{A}(\mathscr{F}_p i + m, \mathscr{F}_p j + n) \tag{1.30}$$

Similarly, average pooling is defined as:

$$M(i,j) = \sum_{m=0}^{c_p} \sum_{n=0}^{h_p} \frac{1}{c_p h_p} \mathscr{A}(\mathscr{F}_p i + m, \mathscr{F}_p j + n) \tag{1.31}$$

1.6 State of the art

1.6.1 Automatic mesh generation

The analysis to generate a compatible mesh that respects the geometric features and the accuracy requirements of numerical solutions takes up to 80 % of the whole meshing procedure on account of automation absence (Hughes et al. (2005)). The automation of the meshing procedure is still considered a critical bottle neck. In recent years, several strategies have been proposed to lessen user intervention to generate good quality meshes. In real world applications, the meshing of input surfaces can be proven a complicated problem as they may contain small gaps, self-intersections etc. There is no robust and automatic way to cleanup a surface. Users have to perform a cleanup manually to assure a well-defined input that can be proven to be a computational laboring work. If the input is not well defined a raw implementation of a meshing algorithm could either not be robust or generate a mesh lacking features of the original geometry. To this extend, an automation aspect of the mesh generation procedure includes techniques to robustly generate high quality meshes of the input geometry and preserve its features as much as possible even if the input is not well defined.

In Hu et al. (2018), a pipeline method is presented to generate tetrahedral meshes taking as input triangle surfaces that represent the boundary of a geometry without connectivity information, also known as triangle soups. Initially, a bounding box that encloses the vertices of the input triangle surfaces is constructed and a Delaunay tetrahedralization is performed. To conform to the input boundary and resolve potential self-intersections of the input geometry, a binary space partitioning is perfomed to detect convex polygons formed by the intersection between the elements of the initial mesh and the triangle faces of the input geometry. The vertices of the polygons are connected with their barycenter to form an initial volumetric mesh within a bounding box larger than the triangle soup. Mesh improvement operations are followed to improve the quality of the mesh. Finally, the winding number filtering algorithm is applied to remove all the elements outside the boundary of the input geometry. In a similar automation context, Guo et al. (2019) propose an automatic triangular surface mesh generation pipeline for CAD models comprised of surface patches. An initial simple coarse triangulation is performed which tessellates the CAD model by preserving the geometric fidelity. This coarse triangulation may include inverted or self-intersecting elements caused by the intersection between the inner and outer boundaries or when a point of the patch is too close to a curve of the geometry. Therefore, a procedure is followed to delete invalid edges and to insert points to avoid an invalid topology. Next, based on the initial triangulation, each patch is meshed in a parametric 2D domain using Constrained Delaunay Triangulation and, then, is mapped back to the 3D object space. Finally, a mesh improvement routine is applied to improve the quality of the mesh.

Automation can also refer to either the inclusion of additional features or the modification to the body of an established mesh generation algorithm. For example, an automated meshing process is presented in Ma & Sun (2019) based on the advancing front method for the creation of quadratic elements. To address the issue of manual domain decomposition for mesh generation with internal feature constraints, the algorithm is enriched with an automatic subregion decomposition that is defined by constraint lines and points in the interior of a geometry and meshes each subregion separately. The extension of the quad-tree algorithm for the creation of quadrilateral elements is addressed in Pochet et al. (2016), where for a better mesh adaption, points of cells near to target curve geometry are attracted to the curve. Next, for a better approximation of the geometry, the cells near the curve are subdivided using six refinement cells instead of four.

Other automatic procedures focus on generating meshes that are suitable computational domains to apply numerical methods. For example, starting from a geometry represented by a stereolithographic (STL) model, Liu et al. (2017) propose an automatic routine that generates polyhedral meshes suitable for the application of scaled boundary finite method to perform stress analysis. An octree grid is created to enclose the geometry and then the mesh is constructed by applying trimming operations that take into account the recovery of sharp features first to the edges, then faces and finally cells. The trimming operations are based on signs assigned to the vertices of the grid that represent whether they are located inside, outside or on the geometry.

The aforementioned work names only a few of the latest advancements on the automation of mesh generation. Although these methods are proven to be efficient and are a step forward to counter explicit user handling, they still might involve the development of complicated algorithms that may include an extensive computational time trade-off.

1.6.2 Mesh Improvement algorithms

Mesh improvement algorithms can either have a global approach or local approach to improve the quality. Global approaches (Alliez et al. (2005), Huang & Russell (2010), Huang (2001)), also known as variational, rely on the minimization of a mesh functional over the domain of the mesh. The vertices of a mesh are moved and the connectivity is adjusted to minimize the mesh functional. The mesh functionals (usually convex) are defined such that the minimization leads to a global minimum. However, apart from the computational cost of such an approach, in many contexts a global dislocation of the vertices that may also not conform to the boundary of the original mesh is not desired. This is more evident in the case of dynamic simulations using Lagrangian meshes. Although global mesh improvement schemes can be applied for simulations using the Lagrangian approach (Bargteil et al. (2007),Wojtan & Turk (2008)), they will also cause the accumulation of numerical errors by smoothing the field values by repeated interpolation on the new location of the vertices. Local approaches on the other hand offer a more explicit handling to mesh improvement by improving the quality using local smoothing, local transformations that replace small groups of tetrahedra with other tetrahedra of better quality, and a scheme that searches for opportunities to apply them. However, local optimization techniques often use highly non-convex functionals and get easily stuck in local minima. As a counter measure, the development of various local mesh improvement schemes and local mesh improvement operations is studied.

In Freitag & Ollivier-Gooch (1997b), a local tetrahedral mesh improvement scheme is presented based on smoothing and topological operations. The smoothing operation moves the vertices to a direction indicated by the solution of a non smooth optimization problem which is acquired using a gradient descent method. The topological operations used are flipping operations along with edge removal. The recommended scheme is to perform two passes of smoothing to the low quality elements followed by a procedure of edge removal to remove the bad tetrahedra, and then perform another two passes of smoothing operation. In Klingner & Shewchuk (2008), the *Stellar* improvement scheme is presented (Section 1.3, *Alg.* 1). This scheme was based on Freitag & Ollivier-Gooch (1997b) with the modification and addition of mesh improvement operations. A constrained vertex smoothing is adapted to lie on flat boundaries. Edge removal was enhanced to be applied at edges that belonged to the boundary of the mesh. Compound operations along with vertex cavitation (referred as vertex insertion), and multiface removal are also added. The same author also presents the *Pulsar* improvement scheme for dynamic simulations. The *Pulsar* scheme prioritizes topological operations and proceeds only if needed to operations that move the original location of vertices (e.g smoothing) to avoid numerical error and artificial diffusion. For boundaries that approximate curved surfaces, quadric smoothing is used that is based on the optimization of an isosurface defined by the distance of a point to the planes created by the neighbor facets; this type of smoothing aims at perturbing the surface as little as possible. The scheme is used by Wicke et al. (2010) to examine the potential of dynamic local mesh improvement of Lagrangian meshes for elastoplastic simulations. To retain the original shape of purely plastic regions, local mesh improvement takes place in a material space that is free of plastic deformations and mapped to the world space. In Clausen et al. (2013), the *Pulsar* scheme is augmented with algorithms for simulating liquids and solid-liquid interaction, merging, and splitting meshes. The boundary domain of the mesh is represented by an algebraic point set surface method (Guennebaud & Gross (2007)) that uses fitting spheres to reconstruct a piece-wise smooth surface. Instead of quadric smoothing, the boundary vertices are allowed to be moved with respect to the reconstructed surface. The improvement scheme is also augmented with face and tetrahedron contraction.

The Longest Edge Propagation Path (LEPP) algorithm (Rivara (1997)) is a vertex insertion strategy based on an ordered sequence of insertions into edges that don't degrade the initial mesh quality. The LEPP path is an ordered path of elements such that each element is adjacent by edge with the previous one and contains another edge of bigger length. Starting with the element containing the largest edge, vertices are inserted in the middle of the adjacent edges to all elements of a path. In 2D, the algorithm can also be used to conduct incremental Delaunay triangulation that will improve the quality of the mesh. The extension to 3D tetrahedral meshes of the algorithm that improves the quality of the mesh by means of refinement is proposed by Rodriguez & Maria (2017).

In Chen et al. (2017) the operation of shell transformation is introduced and included in a tetrahedral mesh improvement scheme. During this operation, the polygon of the ring vertices, referred as shell, including the target edge can be partially triangulated instead of fully triangulated as in the case of edge removal leading to an unmeshed part where faces attached to the edge may still exist. Given one of the remaining faces linked to the target edge due to the partial triangulation of the shell, the operation can be called recursively to be applied to one of the remaining edges of the face to triangulate the edge's corresponding shell. Essentially, the recursive process renders shell transformation a composite edge removal operation that is applied to larger group of tetrahedra than that of edge removal. However, there is no guarantee that a mesh edge is removed by recursively calling the operation under the condition that the mesh quality does not decrease in the process. Therefore, as the operation can be computationally costly a user defined number of recursive levels defines the termination criteria. The operation is used in Zheng et al. (2016) as part local mesh improvement scheme to study moving boundary problems. During this scheme, the polyhedral contour that includes neighbor elements of a bad quality element is extracted. Elements may be added artificially to ensure the topological validity of the polyhedral contour. The polyhedral contour is then meshed using a Delaunay tetrahedralization algorithm with a refinement strategy. After the mesh generation, a mesh improvement scheme that includes shell transformation is used. A parallel version of the scheme is later introduced (He et al. (2019)) using a domain decomposition (Zhao et al. (2015)) enabling to apply the scheme simultaneously to multiple polyhedral contours.

The edge removal operation can be viewed as a combination of 2-3 flips around an edge followed by a 3-2 flip. Dassi et al. (2018) introduce the lazy search flips for an isotropic tetrahedral mesh improvement scheme that can be viewed as a dynamic approach to attempt an edge removal operation. During the operation, a sequence of 2-3 flips are performed around an edge. If the sequence does not improve the mesh quality, then the algorithm reverses the sequence and explores another one. If the quality is improved, no other sequence is explored. If the sequence leads to a configuration where no other 2-3 flips can be performed then the edge is removed by applying a 3-2 flip. The lazy search flips along with global smoothing, insertion and contraction are included in a mesh improvement scheme. Boundary vertices are projected into a surface which is reconstructed by the discrete boundary surface using radial basis functions. When compared to the *Stellar* improvement scheme, better results are achieved in term of mesh quality however more sophisticated methods are required to improve meshes with more complicated curved boundaries.

In Liu et al. (2009), the Small Polyhedron Reconstruction (SPR) operation is presented. A large polyhedral cavity is created around a bad quality element typically comprised of 20-40 elements. The SPR algorithm then performs an exhaustive search of all possible connections to find an optimal tetrahedralization (triangulation in 2D). Compared to flip-based operations that are usually restricted to a local area of the mesh, SPR can achieve better mesh improvement by remeshing larger areas; however, the nature of the operation entails a large computational cost (factorial complexity with regard to number of the polyhedron's vertices). Although modifications of the operation can decrease the time performance of the operation (Liu et al. (2009), Chen & Yang (2014)), it can result in time consuming meshing improvement schemes if the local reconnection is solely based on SPR. Recently, Marot & Remacle (2020) present a mesh improvement scheme that includes a modified version of SPR, Growing SPR, to reduce the computational cost of the operation by addressing the issue of the cavity's formation around bad quality element. Starting from the target element, vertex neighbor elements are added incrementally (with an element quality criterion) to apply the SPR operation until a better tetrahedralization (or triangulation) is found.

1.6.3 Machine learning and meshes

In recent years, advancements in NNs have led to outstanding performances when applied for tasks of object classification and semantic segmentation using images (Simonyan & Zisserman (2015), Sermanet et al. (2014), Chen et al. (2016), ElAdel et al. (2017), Calisto & Lai-Yuen (2020)). Convolution and pooling layers are able to take advantage of the Euclidean regular grid like structure of images to extract local features and offer an invariant framework to variations of an input (LeCun (2012), Krizhevsky et al. (2012)). Recently, there has been an increasing interest to generalize deep learning methods to non-Euclidean structured data such as graphs and manifolds, with a variety of applications from the domains of network analysis, computational social science, or computer graphics. Geometric deep learning (Bronstein et al. (2017)) refers to the field of deep learning applied to non-Euclidean data such as graphs or discrete manifolds (i.e meshes). To generalize convolution in graphs a spectral approach is considered; Observing that the complex exponential corresponds to the eigenfunctions of the Laplacian operator in Euclidean domains, the eigenfunctions of a graph Laplacian operator are considered as a generalized version of the typical Fourier basis. Graph convolution (Bruna et al. (2014), Henaff et al. (2015), Defferrard et al. (2016), Kipf & Welling (2017)) can be achieved by projecting a provided signal to the eigenfunctions of the graph Laplacian operator (graph Fourier transform), multiplying the obtained spectrum with a set of learnable spectral filter coefficients and projecting everything back to the original domain. This intuition has led to good results for signals defined over one graph. However, graph Laplacian eigenfunctions are inconsistent across different domains (basis-dependent). To extend convolution in a consistent way across different domains spatial approaches suggest the application of filters to local patch operators that are intrinsic to a mesh. Such patch operators include the use of geodesic polar coordinates (Masci et al. (2015)), anisotropic heat kernels (Boscaini et al. (2016)), a family of learnable mixture gaussian kernels (Monti et al. (2017)). Other approaches to adapt NN architectures to mesh topological processing suggest the mapping of a mesh to a flat torus topology (Maron et al. (2017)); 2D convolution operators are well defined over a grid of the flat torus. In Verma et al. (2018), the authors propose a dynamic approach to convolution since the operator is calculated according to the features of a vertex. Tangent convolution (Tatarchenko et al. (2018)) projects the local surface geometry on a tangent plane of the vertices of a mesh yielding a set of tangential images that can be treated as 2D grids upon which the convolution operator can be applied. MeshCNN (Hanocka et al. (2019)) is an NN architecture with convolution, pooling and unpooling layers that are adapted to take into account the properties of triangular surfaces. Using as input invariant descriptors of the edges of a mesh, pooling and unpooling operations are defined to collapse edges for the task of mesh simplification. The various adaptions of the convolution operator achieve to efficiently extract features from the meshed geometries assuming however that connections to form elements are already established and therefore such CNN architectures do not address the mesh generation task.

Through the aforementioned advancements of integrating mesh topology to NN architectures tasks such as mesh classification, mesh segmentation and shape correspondence are proven to be efficiently handled by NNs which in turn opened the path for extended applications. In Baqué et al. (2018), given a mesh, Geodesic Convolutional Neural Networks (GCNN) (Monti et al. (2017)) are trained to emulate fluid dynamics simulations by regressing the vector and scalar field values (e.g pressure and drag) over the discrete domain. The GCCN can also be used in an objective function to optimize the mesh representation of shape using an ADAM algorithm with respect to a desirable physics effect. In Wang et al. (2018) the Pixel2Mesh architecture takes as an input a 2D RGB image to generate a mesh representing the depicted object. The architecture combines the classic CNN networks to extract feature from the image with a Graph Convolutional Network (GCN) (Defferrard et al. (2016)) to deform an initial coarse ellipsoid mesh with triangular faces. The vertices of the mesh are adjusted using the GCN while a graph unpooling layer inserts new vertices to the edges of the mesh to refine regions according to the geometrical features of the input image. The architecture is later on extended for generating meshes from multi-view images (Wen et al. (2019)). Litany et al. (2018) use the local spatial patch operators of Monti et al. (2017) to introduce a variational autoencoder that performs shape completion. Although these recent advancements show a promising path for mesh related tasks, they assume the existence of an initial mesh that can be given as input.

The integration of NNs to the mesh generation procedure has been previously studied in both an unsupervised and supervised learning setting. Self-organizing maps (SOM) (Kohonen (2013), López-Rubio & Ramos (2014), Fort (2006)) are unsupervised learning NNs that map multidimensional data onto lower dimensional subspaces where geometric relationships between linked neurons indicate their similarity. The weights of the neurons are adjusted using a competitive learning algorithm; at each iteration, the neurons compete each other to win an input pattern and only one neuron is activated and declared as a winner (Best Matching Unit). The weights of the neurons that are linked with the winner neuron are updated to better fit the input pattern while the weights of the other neurons remain unchanged. SOMs have been utilized for generating meshes based on input patterns that correspond to a set of points distributed inside a geometry by a density function (Ahn Chang-Hoi et al. (1991), Manevitz et al. (1997), Nechaeva (2006)). The density function dictates which parts of the geometry should be approximated by more elements than elsewhere. In the meshing context, a SOM is a fixed grid (triangular or quadrilateral) of linked neurons that adapt their point coordinates (weights) according to the points of the mesh density function using competitive learning. A main drawback of using SOM is that the size of the used grid is fixed which may cause the appearance of badly shaped elements for irregular mesh density functions. Moreover, strategies have to be adopted to ensure that the generated mesh fits the boundary and that elements do not appear outside the boundary of non-convex geometries. In order for the SOM to fit the boundary of the geometry domain Manevitz et al. (1997) suggest a interweaving algorithm of 1D and 2D competitive learning between boundary neurons and interior neurons. For non convex geometries, elements that are located out of the boundary are simply deleted. Based on the interweaving algorithm, Nechaeva (2006) present an improved algorithm that is parralelizable and adapts better on the boundary of non convex geometries.

To counter the drawback of the SOM's fixed size grids Let-it-grow (LIG) neural networks (Alfonzetti et al. (1996) , Triantafyllidis & Labridis (2002)) are suggested for mesh generation. LIG networks start with an initial coarse grid and additional neurons are added in accordance to a density function. During the first phase of the mesh generation algorithm using LIG networks, using competitive learning like SOMs, a point (input pattern) is provided by the density function, the BMU is located and is moved along with its connected neurons towards the position of the point. Next, a signal counter that is assigned to the BMU is incremented and the mesh is checked for topological validity. This is repeated for a specified number of iterations. During the second phase, a new node is added in the midpoint of the edge that contains the neuron with the maximum signal counter and the furthest neighborhood neuron. During the learning process the Delaunay criterion can be used to adjust the connections between the neurons. The algorithms iterate until a desirable number of nodes is achieved. One main drawback of LIG networks lies in the computational complexity of finding the BMU when the mesh includes a large number of neurons. Triantafyllidis & Labridis (2002) suggest an initial Constrained Delaunay Triangulation applied to the boundary of the geometry to form the initial grid upon which LIG competitive learning is applied. Moreover, algorithms that reduce the computational complexity of finding the BMU are presented for the generation of large meshes.

The Growing Neural Gas (GNG) network model also referred as Topology learning network (Fritzke (1995), Martinetz & Schulten (1994)) is an unsupervised learning algorithm that uses competitive learning but unlike SOM and LIG networks does not need an initial specification on the number of neurons or prefixed connections; additional neurons are added and the connections are adjusted as long as the algorithm keeps running. The algorithm starts by adding an initial input pattern and generate neurons that are connected by an edge. The BMU is moved closer to the input pattern as well as all the neurons connected to the BMU. Next, the second BMU (SBMU) is determined. If the BMU and SBMU are connected the age of the edge is set to zero otherwise the neurons are connected. The age of edges emanating from the BMU is then incremented. If an edge has an age larger than a maximum age threshold then it is deleted. If the deletion of the edge results in neurons with no edges then they are deleted as well. After a specified number of iterations, a neuron is inserted halfway between the edges of the worst matching unit and the neighbor neurons. The process is iterated until a specified condition is met, such as a maximum number of iterations. The method is proven to be able to create Delaunay triangulations (Martinetz & Schulten (1994)) under a proper distribution of training input patterns. A raw implementation of GNN to a point cloud however may not result in a topologically valid mesh; after the termination of the algorithm, the set of neurons and edges may not result in global triangle coverage. Therefore, to acquire a triangular mesh, post processing steps are required like removing invalid edges that are not adjacent to two triangular elements, face reconstruction, face reorientation, and fill potential cavities with elements (Holdstein & Fischer (2008), Melato et al. (2007)).

Using supervised learning, in Yao et al. (2005), a NN is utilized to accommodate the meshing process with 2-D quadrilateral elements using the advancing front method. The NN is used to bypass heuristic 'if-then' rules of the element extraction process that define a good quality element. The coordinates of some boundary points are the input of the NN and the parameters that are used to create good quality quadrilateral elements are the output. The construction of training samples relies on manually determining the patterns of good quality quadrilateral elements. The NN is able to extract good quality elements, however, it is restricted to be trained with a limited number of boundary points as the complexity to find training patterns increases with the increase in the number of boundary points that are included as the input to the NN.

In Vinyals et al. (2015), pointer networks are introduced as a new neural architecture. The networks consist of an encoding and a decoding Recursive NN (RNN). At each step of the decoding process, a pointer selects a member of the input sequence as the output. Unlike sequence-to-sequence models (Sutskever et al. (2014)) and Neural Turing Machines (Graves et al. (2014)), the size of the output does not need to be fixed a priori. Pointer networks are applied to explore the application of NNs for combinatorial problems where the size of the output is variable and depends on the size of the input. The problem of Delaunay triangulation is examined on a set of points using this architecture. The inputs are the coordinates of the points and the output is a set that contains triplets of integers that correspond to the order of the input and represent the vertices forming the triangles. As the network is not directly addressed for mesh generation, the resulting meshes may have intersecting connections and partial triangle coverage.

A recent addition for mesh generation with NNs using supervised learning is the MeshingNet NN (Zhang et al. (2020)) that guides standard mesh software to generate meshes with an element distribution that provides accurate solutions when solving Partial Differential Equations (PDE) with FEM. The traditional approach to generate such a mesh involves an a posteriori error estimation. An initial solution is calculated on a relatively coarse mesh and then auxiliary problems are calculated on an element or a patch of elements to approximate local errors. The local errors can be used in conjunction with the initial solution to approximate a global error that can indicate which regions of the mesh should either contain more elements (refinement) or potentially contain less elements while not harming the accuracy of the solution. MeshingNet takes as input the coordinates of the geometry's boundary (polygon), parameters of the PDE and mean value coordinates of a point (parametric coordinates relative to the vertices of a polygon) inside the geometry and outputs the local area upper bound around that point that dictates the local element distribution. To train the network, the local area upper bound is calculated based on the error on solving the PDE on a low density uniform element mesh and a high density uniform element mesh. After the training, a low density uniform mesh is generated for the given geometry and the vertices of the mesh are given as input to the network. Based on the local area upper bound of the inner vertices, a mesher is called to refine accordingly.

1.7 Research Objective

The goal of the present research work is to explore the potential of machine learning techniques applied to a mesh generation and local mesh improvement framework. In particular, NNs that are proven to be effective for problems that are complex and time consuming are integrated in an automatic mesh generation and mesh improvement procedure. Despite the variety of previous approaches in the use of machine learning frameworks for mesh generation and improvement, none of these methods can currently replace the standard algorithms which are mostly based on grid based, constrained Delaunay or advancing front methods. The integration of machine learning in the field of mesh generation and mesh improvement remains an area open to exploration with great potential. The main scope of this thesis is therefore to explore the integration of NNs and study their accuracy when used as a main component for: (i) robust mesh generation and (ii) mesh improvement. The intermediate steps to accomplish this goal are categorized into the development of the following techniques:

1. Mesh generation for small contours

A novel data driven mesh generation framework using NNs is proposed for simplicial contours. The contours have a maximum number of $N_C = 16$ edges. The presented meshing scheme uses NNs to generate triangular meshes of good quality on 2D contours for a target element size. Each of the NNs is assigned to predict a step towards mesh generation. Three NNs are used to predict the number of inner vertices that must be inserted inside the cavity of the contour, their location, and how to connect vertices to form elements. The predictions of a NN assigned to predict a step of the meshing scheme are pipelined to proceed to the next step. The meshing scheme generates meshes, based on datasets of meshed contours that are generated using a reference 2D CDT meshing algorithm. The accuracy of the scheme is evaluated by comparing the quality of the mesh generated by the neural networks with that generated by the reference mesher.

2. Local mesh improvement operations

The trained NNs of the aforementioned meshing scheme are then used to develop local mesh improvement operations with the addition of NNs that reposition vertices of the mesh to improve its quality. First, the mesh is partitioned into local mesh configurations. The local mesh configurations contain low quality elements or edges that are either too short or long according to user defined target edge lengths. The contours of the local mesh configurations are extracted and a mesh improvement operation is applied to them. The number of the contours edges is in accordance with the trained NNs of the meshing scheme. The developed operations are validated and evaluated in terms of the quality outcome as part of local mesh improvement schemes that are applied for static meshes and dynamic meshes where the vertices move according to a prescribed equation.

3. Large mesh generation

Finally to overcome the limitation of the meshing scheme's application for a limited number of contour edges, an extended scheme is presented for the generation of large size meshes. The scheme is based on the application of all the trained NNs used in the aforementioned procedures. Given a high resolution contour that represents the boundary of a geometry, vertices are sampled to form a low resolution contour with a number of edges that conforms to the trained NNs. The low resolution contour is meshed using the initial meshing scheme. The mesh is then refined by inserting vertices on the elements edges. The inserted vertices that belong to the edges of the low resolution contour are projected to the high resolution contour. Additional vertices are added in the interior of the sub-contours that are formed through this process and meshed using the NNs of the meshing scheme for small contours. To further improve the quality of the reproduced mesh, the NNs that reposition the vertices are called. The extended scheme is adapted to generate meshes with both a uniform and adaptive scale element size.

1.8 Outline

Apart from the introduction, the rest of the thesis is organized as follows:

In **chapter 2** the mesh generation scheme for small contours is described. The problem statement along with the pre-processing steps of training data acquisition and feature transformation of the contour population are initially described. Subsequently, the methodology and NN architecture for each step of the meshing scheme is provided. These steps include: (i) the prediction on the number of inner vertices to be inserted inside the cavity of a contour based on a target edge length, (ii) the prediction of the location of the inner vertices, and (iii) the prediction of the connectivity to form the elements of the mesh. Because the training of the NN that predicts the connectivity is based on sampling vertices inside a contour, an adaptive sampling strategy is presented to reduce the population of training data. **Chapter 3** contains the results of applying the meshing scheme on random contours. The error metrics along with the experimental parameters of the training population sizes, NN hyper-parameters, and computational resources are presented. The accuracy of each step and that of the overall meshing scheme are evaluated. Results on the efficiency of the inner vertices adaptive sampling strategy to reduce the training population are also presented.

Chapter 4 contains the overview of the mesh improvement operations that are developed based on the trained NNs of the meshing scheme. These operations include: (i) reconnection, (ii) vertex repositioning, (iii) surface control , and (iv) size control. A description of the vertex repositioning and surface control NNs is also provided. In **chapter 5**, the mesh improvement operations are included in local mesh improvement schemes to validate and evaluate their efficiency. The training populations and NN hyper-parameters are provided.Tests are carried out on static and dynamic meshes. The vertices of static meshes are perturbed and their edges are randomly swapped to degrade their quality. The quality and angle distribution of the elements before and after the application of the local mesh improvement scheme is presented. The efficiency in dynamic meshes where the vertices move according to an analytical velocity field is measured in terms of minimum quality of the mesh before and after the application of the local mesh improvement scheme for each simulation time step.

In **Chapter 6** the extension of the meshing scheme for the generation of larger meshes is described. The algorithms for uniform and adaptive element scale size are presented. Examples of meshed geometries with their corresponding qualities are also demonstrated.

Chapter 7 concludes the thesis by offering a final summary of results and conclusions. A general outlook and objectives on future research are also provided.

2 Meshing of 2-D simplicial contours using Neural Networks

2.1 Problem Statement

In this chapter, the problem of simplicial mesh generation for a target element size is studied in bounded domains. The boundary of the domain ∂V, i.e contour, is composed of piecewise linear segments forming the edges of the contour (Fig. 2.1). The continuous interior domain V, i.e cavity, is then tessellated into $V_i, i = 1, 2, ...N_{el}$ triangular elements. To achieve a good quality mesh composed of a target element edge size (target edge length), inner vertices are strategically placed in the cavity of the contour. The final mesh is topologically valid if each element edge is incident to only one or two elements and there are no element entanglements (manifold condition).

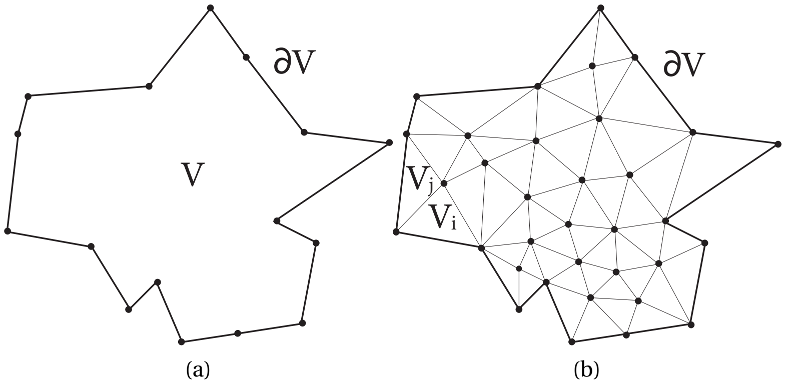


Figure 2.1: (a) A set of points and edges defines a closed boundary ∂V (contour) with an interior continuous domain V (cavity). (b) To form a good quality mesh consisting of simplicial (triangular) elements whose edges respect a specific length, vertices are inserted in strategic locations of the interior domain V and are connected resulting in the discretization of V into $V_i, i = 1, 2, ...N_{el}$ triangular elements, where N_{el} is the number of elements such that $\cup_{i=1}^{N_{el}} V_i = V$. The intersection of sub domains $V_i \cap V_j$ is at most an element edge; the tessellated domain does not contain intersections between the elements of the mesh.

2.2 Algorithm overview

The machine learning methods that are used for data with a grid-like underlying structure (e.g image processing) can't be applied to a meshing framework. Unlike the grid structure of pixel-based data, the underlying Euclidean space of possible contour inputs does not have such an organized structure. Due to the lack of structure, a naive brute force method of providing as input the unprocessed point coordinates of a random contour does not result in sufficiently robust and accurate pattern recognition from the NNs (Yao et al. (2005)). The proposed scheme is instead intentionally designed to ensure a high level of consistency among the data provided to the NNs such as to ease NNs learning abilities while minimizing the amount of learning data. To this end, the first stage of the presented algorithm consists of a pre-processing step that applies a feature transformation to best fit a reference contour circumscribed in a unitary circle. This transformation provides a scale and rotation invariant shape representation of the contours. Furthermore, the contour vertex indexing has to follow a specific ordering rule, e.g clockwise or anti-clockwise, to underline patterns of vertex connections that form the edges of the contour.

To achieve a target element size, the target edge length is part of the inputs of the NNs of the scheme. The accuracy of an approximation of the inner vertices from a reference mesher is very crucial to the overall mesh output of the presented meshing scheme. The use of a grid covering the contour is opted to estimate the location of the inner vertices. The cells of a grid are associated with a distance function. The location of the inner vertices is stored in the form of a distance function to these vertices. A local interpolation is performed around grid points to maximize the accuracy of the approximation.

To avoid intersection between elements and opt for connections that lead to good quality elements, a triangulation algorithm is used. The triangulation algorithm is based on the output values of a NN. The output values of the NN represent the probability of connecting the face of an element (edge) to another vertex of the mesh. The values are stored in a connection table which is used by the triangulation algorithm to assess the connections and avoid element entanglement.

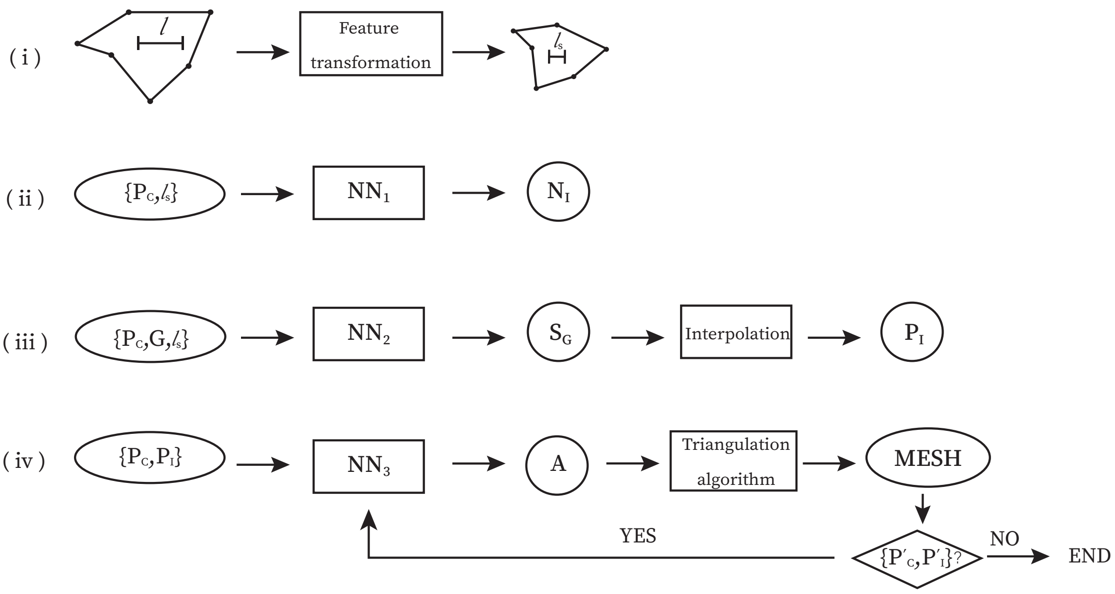


Figure 2.2: The meshing scheme consists of four steps: (i) The initial contour with N_C edges is scaled and rotated with respect to a regular polygon with N_C edges inscribed in a unitary radius circle. (ii) NN_1 is used to approximate the number of inner points N_I, providing as input the contour vertex coordinates P_C and the requested target edge length l_s. (iii) NN_2 takes as input the vertex coordinates P_C, patches of grid points from G, and the target edge length l_s. It outputs the scores S_G for each grid point. Based on S_G, N_I grid points are selected and interpolation is applied to a region around them. Next, to approximate the inner vertices P_I, the local minimum of the interpolated surface are found. (iv) NN_3 takes as input the contour vertices P_C and the inner point vertices P_I and outputs the entries of a connection table A. The contour is meshed with a triangulation algorithm that meshes the cavity of the contour based on A. After the termination of the algorithm, if a sub-contour with $P_C^{'}$ contour coordinates is created containing $N_I^{'}$ inner vertices with $P_I^{'}$ coordinates, NN_3 is called recursively to mesh the sub-contour, until no further sub-contour emerges.

The proposed meshing algorithm consists of the following steps (Fig. 2.2, Fig.2.3):

(i) To mesh a contour of N_C edges for a user target element size of edge length l, first a feature transformation is applied to it. The contour is scaled and rotated with respect to a regular polygon of N_C edges that is inscribed in a circle of unit radius. This transformation also changes l to $l_s = Sl$, where S is the scaling factor of the transformation and l_s is the scaled target edge length.

(ii) The first neural network NN_1 takes as input the transformed contour vertex coordinates $P_C = \{p_i = (x_i, y_i), i = 1, 2, ...N_C\}$ of the transformed contour and target edge length l_s. The output of NN_1 is the number of vertices N_I that should be inserted inside the cavity of the contour, i.e the interior domain of the contour, to achieve the target edge length l_s.

(iii) The approximation of the coordinates of the inner vertices $P_I = \{p_{I,i}, i = 1, 2, .., N_I\}$ is done with the help of a square grid G over the contour. The second neural network NN_2 takes as input the coordinates of the contour P_C, the target edge length l_s, and patches of G. A surface is defined over the grid whose local minima determine the most probable locations of the inner vertices. NN_2 outputs the values of the surface for the grid points contained in the input patch. Based on these values, N_I grid points are selected and a regional interpolation follows. Finally, the local minima of the interpolated surface reveals the approximated locations of P_I.

(iv) The third neural network NN_3 takes as input the vertex coordinates P_C of the transformed contour and the coordinates of the inner vertices P_I and outputs the entries of a connection table A. The connection table contains values representing the probability of connecting the face of an element (edge) to one of the inner vertices $p_{I,i}$ or with another contour vertex p_i. The mesh is created using a triangulation algorithm that is based on the values of the connection table. Because the triangulation algorithm connects contour edges with vertices, this may lead to the formation of inner sub-contours that are not meshed. In this case, the triangulation algorithm is called recursively to mesh the sub-contours.

Gmsh© is used as the reference mesher (see **Appendix A.2.1**). *Gmsh*© is a wrapper implementing a Delaunay algorithm, written in *C*++, as presented in Lambrechts et al. (2008). The NNs are implemented and trained using *Pytorch* (Paszke et al. (2017)). The triangulation algorithm used to predict the connection of the mesh has been developed by the author and implemented in *Python*.

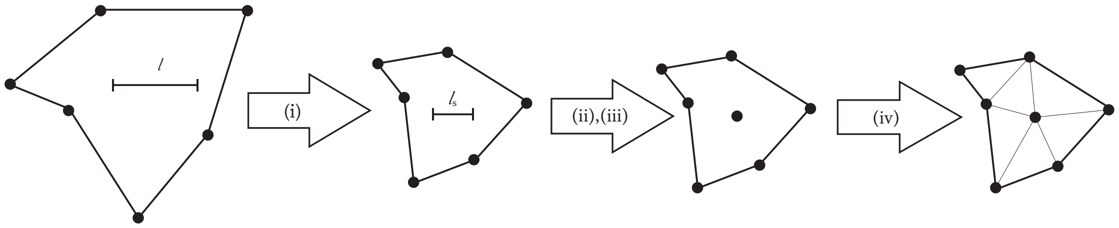


Figure 2.3: From left to right: Steps followed to acquire the mesh of a contour. Step (i) of the algorithm consists of a feature transformation applied to the contour that causes the scaling of the target elements size of edge length l to l_s. Next, following steps (ii) and (iii) of the proposed meshing scheme, based on the prediction of NN_1, one vertex is inserted in the interior of the contour (cavity), and its location is predicted using NN_2. The final step (iv) of the meshing scheme uses NN_3 and a triangulation algorithm to connect the edges of the contour with inner vertices or contour vertices to create the mesh.

2.3 Feature transformation and training data acquisition

The connectivity of a mesh is not changed if a contour is rotated or scaled. The contour to be meshed undergoes a feature transformation that assists pattern recognition by the

NNs involved in the scheme. The required scaling and rotation invariance is achieved by applying the Procrustes superimposition on a reference contour (Gower (1975)). For a contour with N_C edges and P_C^* contour coordinates, a regular polygon is used with N_C edges inscribed in a unit circle as a reference. Procrustes superimposition imposes a linear transformation to the contour points $P_C^* = \{p_i^*, i = 1, 2, ...N_C\}$ so that they best conform to the points of the reference contour $Q_C = \{q_i, i = 1, 2, ...N_C\}$ (Fig. 2.4). This pre-processing step is essential to ensure consistency among the data provided to the NN. The centered Euclidean norms $||P_C^*|| = \sum_{i=1}^{N_C} (p_i^* - \overline{p^*})^2$ and $||Q_C|| = \sum_{i=1}^{N_C} (q_i - \overline{q})^2$, where $\overline{p^*} = \sum_{i=1}^{N_C} p_i^* / N_C$ and $\overline{q} = \sum_{i=1}^{N_C} q_i / N_C$, scale the coordinates of P_C^* and Q_C to the same unit norm by applying the transformation $P_{||C||}^* = P_C^* / ||P_C^*||$ and $Q_{||C||} = Q_C / ||Q_C||$. Singular Value Decomposition (SVD) is applied to $A = Q_{||C||}^{\top} P_{||C||}^*$. SVD decomposes A to $A = UCV$ which yields the optimal rotation matrix $R = UV^{\top}$ and the scaling factor $S = ||Q_C|| tr(C)$. The rotation matrix and the scaling factor define the transformation $F : P_C^* \rightarrow P_C$, $F(P_C^*) = P_C = S(P_C^* / ||P_C^*||)R + \overline{q} = SP_{||C||}^* R + \overline{q}$. Similarly, if $P_I^* = \{p_{I,i}^*, i = 1, 2, ...N_I\}$ are the coordinates of inner vertices of a contour cavity and N_I is the number of inner vertices, $F(P_I^*)$ maps the vertices to their relative locations P_I inside the transformed contour.

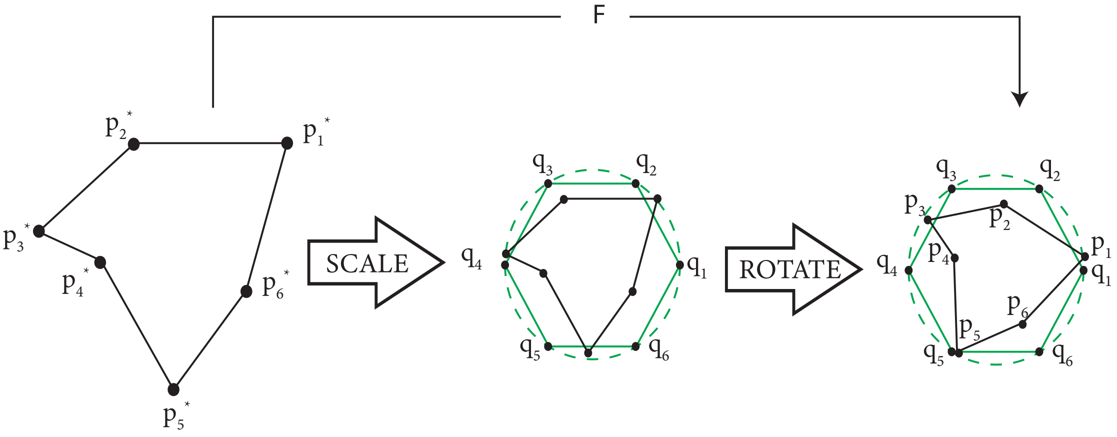


Figure 2.4: Procrustes superimposition on contours with 6 edges with requested element size l. The reference contour is a regular hexagon inscribed in a unit circle with coordinates $Q = \{q_i, i = 1, 2, .., 6\}$. The contour is scaled by a scale factor S, changing the target edge length from l to $l_s = Sl$, and rotated to best fit the point of the reference polygon to acquire the points of the transformed contour $P_C = \{p_i, i = 1, 2, .., 6\}$.

The training of the NNs is based on sets of contours with N_C edges. To generate a mesh contour of N_C edges (Fig. 2.5), random points are chosen from a disc of unit radius that is divided into N_C sectors. A point is randomly selected from each sector and these points are connected subsequently to form a random contour. To avoid the creation of a contour with very short edges, the selection of points is excluded from the inner region of a circle with small radius r.

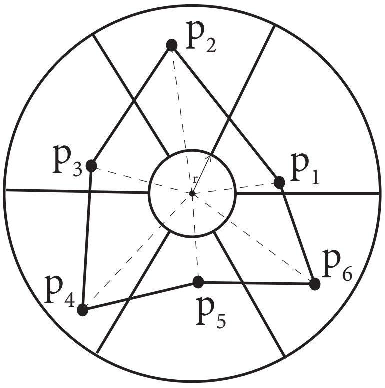


Figure 2.5: Example of creation of contour with 6 edges. A unit circle is divided into 6 sectors. From each sector a point $p_i, i = \{1, .., 6\}$ is selected. To avoid the creation of contours with very short edges, no points are selected from the inner region of a circle with small radius r.

After applying the Procrustes transformation to the contour sets, the contours are meshed using Constrained Delaunay Triangulation (**CDT**) (Paul Chew (1989)) followed by a refinement process for multiple-scaled target edge lengths l_s. CDT generates an initial triangular mesh with respect to the boundaries of a contour. Subsequently, inner vertices (Steiner points) are inserted in the cavity of the contour and the connections are updated to comply with the target edge length and satisfy quality criteria resulting in a graded mesh. Constraints are added to avoid vertex insertion along the edges of the contour. The number of inner vertices N_I inserted and their coordinates P_I are then used to train NN_1 and NN_2 respectively. The information on the inner vertices also allows to compute the connection table A of the contour that is used to train NN_3 (Fig. 2.6).

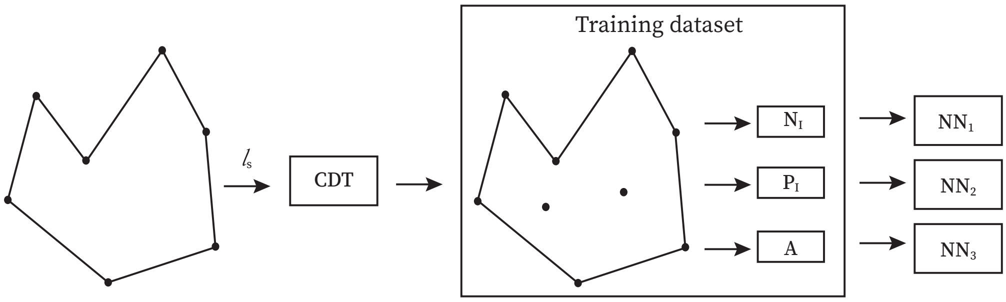


Figure 2.6: The generated contours are meshed by applying CDT followed by refinement for various target edge lengths l_s producing a graded mesh. The number of inner vertices N_I and their coordinates P_I from the graded mesh are used to train the NN_1 and NN_2. By knowing the location P_I of the inner vertices, the connection table A of the contour is calculated to be included in the training dataset of NN_3.

2.4 Prediction of the number of inner vertices

To predict the number N_I of inner vertices NN_1 is used. The N_I inner vertices that are inserted during the refinement process (after creating an initial mesh with CDT) inside a contour is used to train NN_1 which outputs an approximation $\hat{N}_I$. Based upon the approximated number of inner vertices $\hat{N}_I$, the meshing scheme proceeds to approximate their location.

NN_1 is a feedforward NN with multilayer perceptrons. For a contour with N_C edges, NN_1 takes as input the contour vertex coordinates P_C and target edge length l_s (Fig. 2.7). The network is trained to minimize the loss function $\mathscr{L}(N_I, \hat{N}_I) = |N_I - \hat{N}_I|$, where N_I is the number of vertices that are inserted during the refinement process of the reference mesher and $\hat{N}_I$ is the number of vertices that are predicted by NN_1 (Alg. 3).

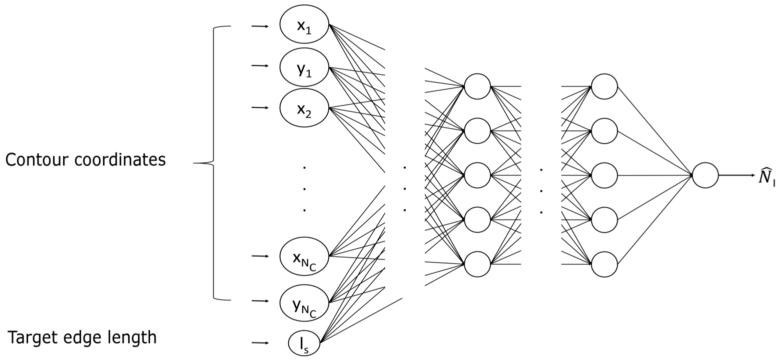


Figure 2.7: NN_1 architecture for the prediction of number of vertices. It takes as input the contour coordinates $P_C = \{p_i = (x_i, y_i), i = 1, 2, .., N_C\}$ and the scaled target edge length l_s. NN_1 outputs the approximation $\hat{N}_I$ of the number of inner vertices that should be inserted inside the cavity of the contour to achieve the target edge length l_s.

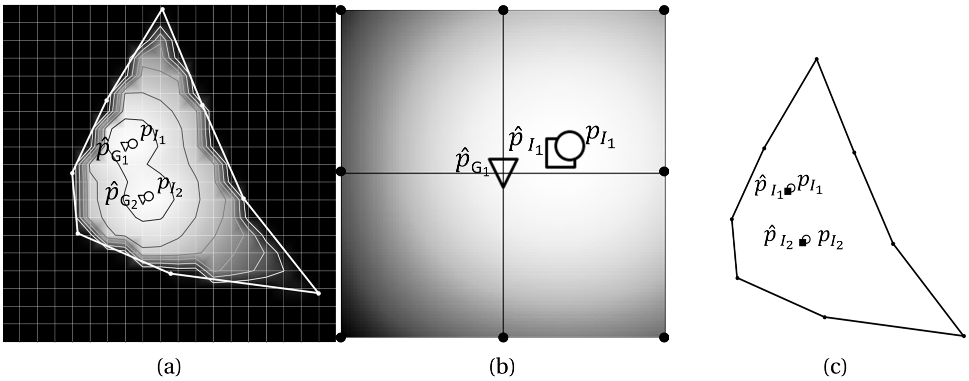
Algorithm 3: Training algorithm of NN_1 for the prediction of number of inner vertices.

```
$P_C$: Contour vertices coordinates
$l_s$: Target edge length
$N_I$: Number of inner vertices
$\hat{N}_I$: Estimated number of inner vertices
$N_{train}$: Number of training data population
Initialise weights $W_K$ of $NN_1$
while required number of iterations is not reached do
    foreach training example in $D = \{(P_C^{(n)}, l_s^{(n)}, N_I^{(n)})\}_{n=1}^{N_{train}}$ do
        Compute $\hat{N}_I^{(n)}$ using current parameters
        Calculate loss function $\mathscr{L}(N_I^{(n)}, \hat{N}_I^{(n)}) = |N_I^{(n)} - \hat{N}_I^{(n)}|$ and $\frac{\partial \mathscr{L}}{\partial W_K}$
        Update $W_K$ using Adam learning rate optimization
    end
end
```

2.5 Prediction of the inner vertices positions

To approximate the location of the inner vertices inside the cavity, a grid G of resolution $N_G = n_G \times n_G$ is first defined inside a boundary box that includes all the vertices P_C of the contour. After acquiring the coordinates of the inner vertices P_I from the graded reference mesh, the distance between each grid point and inner vertex in the contour is calculated. Each grid point is assigned a score, which is defined as the distance to the closest vertex. The scores of the grid points are used to choose those that are closest to the inner vertices (Fig. 2.8a). First, the grid point with the lowest score is selected. Next, interpolation is applied to find the scores on a local domain of the selected grid point to determine the local minimum (Fig. 2.8b). The minimum is the approximation of an inner vertex. The number of grid points included in the local domain depends on the target edge length l_s. Finally, the grid points around the selected grid point are excluded. The same procedure is applied for the next grid point with the lowest score to acquire the next approximation of an inner vertex (Fig. 2.8c).

NN_2 is used to predict the scores of the grid points. The use of the grid allows the adaption of the score of the grid points so that the predictions of NN_2 lead to valid triangulations and adhere to the target edge length. To avoid the predictions of vertices that are located outside the contour, the score of the grid points that are located near or outside the boundary of the contour is penalized. After choosing a grid point with minimum score, the grid points around it are restricted from being selected for the approximation of the subsequent inner vertex, to ensure that the predictions complied with the desired target edge length. This procedure avoids predictions of inner vertices that are too close due to inaccuracy of the scores by NN_2.



(a) (b) (c)

Figure 2.8: Example of the NN_2 approximation of two inner vertices $p_{I,1}$ and $p_{I,2}$ ($N_I = 2$) for a contour with 8 edges. (a) Based on the scores S_G, the grid points $\hat{p}_{G_1}$ and $\hat{p}_{G_2}$ are selected as the first two grid points with the minimum score. (b) Then, interpolation is applied to a local region around them. Here, interpolation is applied to find the scores on a region around $\hat{p}_{G_1}$. This region includes the grid points around $\hat{p}_{G_1}$. The number of grid points included in the region depends on the target edge length l_s. By locating the local minimum of the interpolated surface, the approximation $\hat{p}_{I,1}$ of $p_{I,1}$ is acquired. (c) The interpolation procedure is also applied to $\hat{p}_{G_2}$ to obtain $\hat{p}_{I,2}$.

NN_2 is a feedforward NN with multilayer perceptrons that takes as input the contour vertex coordinates P_C, the coordinates of the grid points P_{G_k} contained in each patch, and the target edge length l_s (Fig. 2.9). It outputs the scores $s_{i,g}$ for the points contained in the patch. The objective loss function is the mean squared error $\mathscr{L}(s_{i,j}, \hat{s}_{i,j}) = \sum_{i,j}(s_{i,j} - \hat{s}_{i,j})^2 / N_{G_k}$, where $s_{i,j}$ is the calculated score of the grid point $p_{G_{i,j}}$, $\hat{s}_{i,j}$ is the score that the NN outputs, and N_{G_k} is the number of grid points included in the patch (Alg. 4).

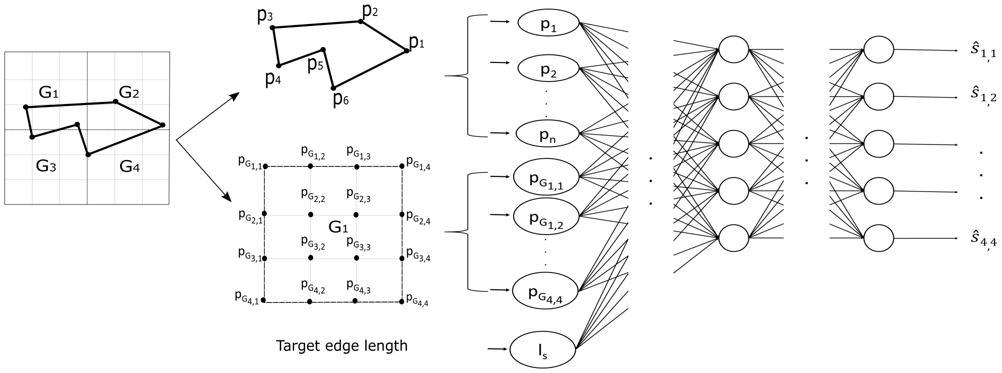


Figure 2.9: The grid G defined over the contour is divided into N_p patches $G_k,\ k = \{1,2..N_p\}$ (here $N_p = 4$). NN_2 takes as input the contour coordinates $P_C = \{p_i, i = 1,2,..,N_C\}$, the coordinates of the N_{G_k} grid points $P_{G_k} = \{p_{G_{(i,j)}} : (i,j) = \{1,2,..,a\} \times \{1,2,..,a\}, a^2 = N_{G_k}\}$ that are included inside a patch, and the target edge length l_s. It outputs the scores $\hat{s}_{i,j}$ that correspond to each grid point inside the patch.

Algorithm 4: Training algorithm of NN_2 for the prediction of location of inner vertices.

```
$P_C$: Contour vertices coordinates
$l_s$: Target edge length
$P_{G_k}$: Coordinates of grid points contained in the patch
$S_{G_k}$: Scores of grid points contained in the patch
$\hat{S}_{G_k}$: Estimated scores of grid points contained in the patch
$N_{train}$: Number of training data population
Initialize weights $W_K$ of $NN_2$
while required number of iterations is not reached do
    foreach training example in $D = \{(P_C^{(n)}, l_s^{(n)}, P_{G_k}^{(n)}, S_{G_k}^{(n)})\}_{n=1}^{N_{train}}$ do
        Compute $\hat{S}_{G_k}^{(n)}$ using current parameters
        Calculate loss function
          $\mathscr{L}(S_{G_k}^{(n)}, \hat{S}_{G_k}^{(n)}) = ||S_{G_k}^{(n)} - \hat{S}_{G_k}^{(n)}||_2^2 / N_{G_k} = \sum_{i,j}(s_{i,j}^{(n)} - \hat{s}_{i,j}^{(n)})^2 / N_{G_k}$ and $\frac{\partial \mathscr{L}}{\partial W_K}$
        Update $W_K$ using Adam learning rate optimization
    end
end
```

2.6 Prediction of the connectivity

2.6.1 Triangulation algorithm

The final step of the meshing algorithm is to find the most probable connections between a face (edge) of the contour and a vertex, that would lead to a good quality mesh. The connection table A is a tool that lists the different probabilities for each contour edge to be connected with either a contour vertex or an inner vertex. Based on the values of A, a triangulation algorithm is used to mesh the contour.

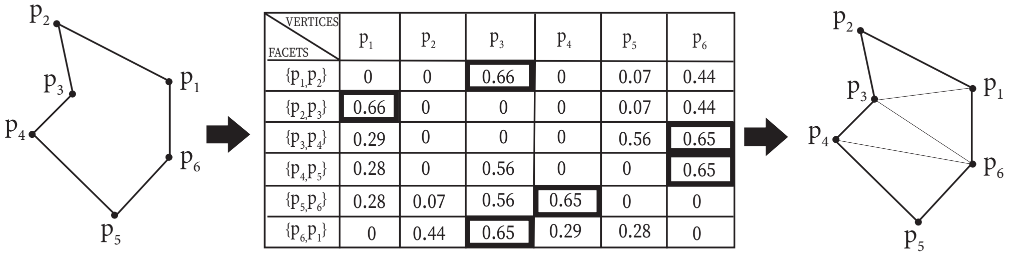
Given a facet F_i (edge in 2-D) of a contour, the probability $P(F_i, v_j)$ of connecting this facet to a vertex v_j to form an element must be determined. For each entry $P(F_i, v_j)$, starting with the contour, the facet F_i is connected to a vertex v_j and then the remaining region is meshed using CDT. The element quality of the resulted mesh is measured using the following metric:

$$q_{el} = \frac{4A_{el}}{\sqrt{3}l_{rms}^2} \tag{2.1}$$

where $0 < q_{el} \leq 1$, A_{el} is the area of the triangular element and $l_{rms} = \sqrt{\frac{1}{3}\sum_{i=1}^{3} l_i^2}$, where $l_i, i = 1,2,3$ are the edge lengths of the triangular element. The lowest and mean element quality, q_{worst} and q_{mean}, respectively, are calculated among the mesh elements, and stored as the probability $P(F_i, v_j) = (q_{worst}, q_{mean})$. q_{mean} is used to differentiate between two vertices of the same lowest quality q_{worst}. If the facet F_i to a vertex v_j forms an elements that is located outside the interior domain of a contour, then the probability is omitted to zero. Such elements are spotted by calculating their signed area. Due to the anti-clock wise ordering of the contour vertices, the signed area of such an invalid element is negative.

The connection table A contains the entries $a_{i,j} = P(F_i, v_j)$, where $a_{i,j}$ is the probability that the F_i (i^{th} row) facet of the contour connects with the v_j (j^{th} column) vertex; v_j being either a contour vertex or an inner vertex. The connection table is then ordered by increasing quality; first the facets are ordered (rows) and then the vertices (columns). Given an ordered connectivity table, a region is meshed by the following procedure (Fig. 2.10): the entry with the highest probability is first chosen. If two entries having the highest probability had the same value of q_{worst}, the one with the highest q_{mean} is favored. This entry indicates which facet is to be connected to which vertex. The row containing the entries for the former facet is then eliminated.

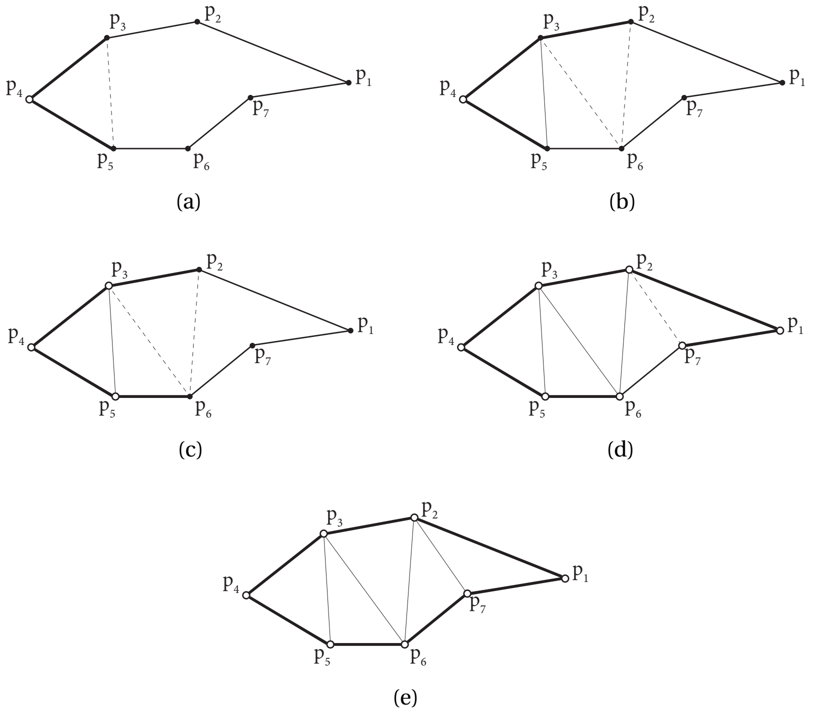
After a facet has been connected, a element is formed that contains vertices and facets which may not be available for further connection. In such a case, the vertices and facets is considered to be locked and are included in a set of locked vertices V_{locked} and a facets are tagged as locked F_{locked} (Fig. 2.11) (Alg. 5, Lines 23-30). The connection of facets in F_{locked} is bypassed by removing the row from the connection table (Alg. 5, Line 27). The connections of facets with vertices included in V_{locked} is also bypassed (Alg. 5, Lines 14-16) (for more details on locking mechanism, see **Appendix A.3.1**). Connections that crossed existing mesh elements



VERTICES / FACETS	p_1	p_2	p_3	p_4	p_5	p_6
$\{p_1,p_2\}$	0	0	$\boxed{0.66}$	0	0.07	0.44
$\{p_2,p_3\}$	$\boxed{0.66}$	0	0	0	0.07	0.44
$\{p_3,p_4\}$	0.29	0	0	0	0.56	$\boxed{0.65}$
$\{p_4,p_5\}$	0.28	0	0.56	0	0	$\boxed{0.65}$
$\{p_5,p_6\}$	0.28	0.07	0.56	$\boxed{0.65}$	0	0
$\{p_6,p_1\}$	0	0.44	$\boxed{0.65}$	0.29	0.28	0

Figure 2.10: The cavity is meshed according to the entries of the connection table. Here, the connection table contains the values of q_{worst}. Each facet (row) is connected with the vertex (column) that has the maximum value of q_{worst}. First, each row is ordered by increasing quality. Subsequently, the columns are ordered with the same criteria. Once the connection table is sorted, the meshing algorithm is called. When computing the connection table, the connection entry of a facet with a vertex that forms a element outside the cavity of contour (e.g $\{p_2, p_3, p_4\}$) is omitted to zero by computing the signed area of the element. Valid elements have a positive signed area, while invalid have a negative signed area. In the depicted example, the triangulation algorithm starts by connecting the facet $\{p_1,p_2\}$ with the vertex p_3 to create the element $\{p_1,p_2,p_3\}$. This connection is done with accordance to the higher value of the row of the connection table (i.e 0.66). The creation of $\{p_1,p_2,p_3\}$ locks the vertex $\{p_2\}$ for any further connections. Since $\{p_2,p_3\}$ is another contour facet included in the formed element, $\{p_2,p_3\}$ is tagged as locked (F_{locked}). This removes the row $\{p_2,p_3\}$ and the algorithm proceeds to connect facet $\{p_2,p_3\}$ with p_6. In a similar fashion, all the facets of the contour are connected with the vertex that contains the highest entry to form the mesh incrementally, one element after the other.

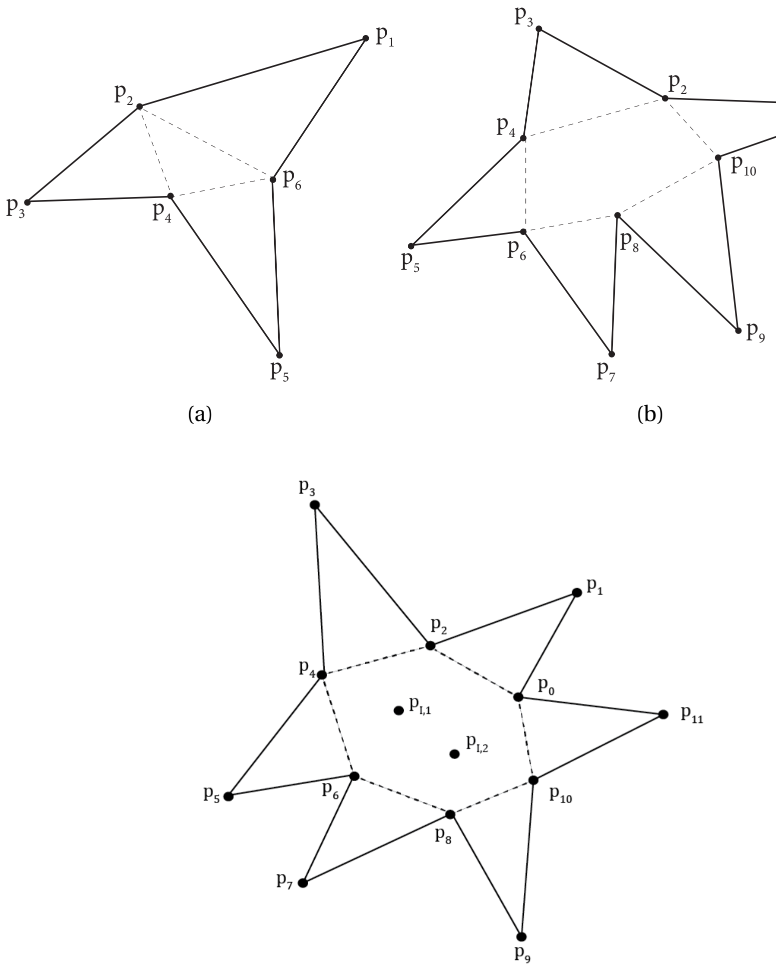
are naturally avoided by assigning a group id to separate sub-cavities (Alg. 5, Lines 19-23) and enforcing connections among facets and vertices having the same group id (Alg. 5, Lines 14-16) (see **Appendix A.3.2** for spotting sub-contours). This procedure is repeated for the next highest entries.



(a) (b) (c) (d) (e)

Figure 2.11: (a)-(e) Example of meshing a 2-D cavity while the sets V_{locked} and F_{locked} are updated. (a) First, the facet (edge) $\{p_3,p_4\}$ is connected to vertex p_5. The creation of the element $\{p_3,p_4,p_5\}$ locks vertex p_4, as it can no longer be connected with another facet. The facet $\{p_4,p_5\}$ is also locked as it can no longer connect with another vertex. V_{locked} now contains vertex p_4 and $\{p_4,p_5\}$ is tagged as F_{locked}. (b) Facet $\{p_2,p_3\}$ connects with vertex p_6 creating the element $\{p_2,p_3,p_6\}$. (c) The creation of element $\{p_2,p_3,p_6\}$ causes the apparition of element $\{p_3,p_5,p_6\}$. V_{locked} will be updated with vertices p_3 and p_5 and facet $\{p_5,p_6\}$ will be tagged as F_{locked}. (d) By connecting the facet $\{p_1,p_2\}$ with vertex p_7 the element $\{p_1,p_2,p_7\}$ locks vertex p_1 and tags facet $\{p_1,p_7\}$ as F_{locked}. (e) The creation of element $\{p_1,p_2,p_7\}$ causes the apparition of element $\{p_2,p_6,p_7\}$ that locks vertices p_2 and p_6. All vertices are now included in V_{locked} (termination of the algorithm)

At the end of the procedure vertices that remain open for connection may cause the appearance of sub-cavities. If the sub-cavity contains only a single element (Fig. 2.12a), this element is merely added to the mesh. Otherwise, the sub-cavity forms a sub-contour that may also contain inner vertices (Fig. 2.12b, 2.12c). In such a case, the algorithm is called recursively (Alg. 5, Lines 32-37).



(a)

(b)

(c)

Figure 2.12: (a) Example of the appearance of a sub-cavity with a single element. After a first iteration of the triangulation algorithm the facets of the contour are connected with the vertex that corresponds to the highest entry of the connection table. This also causes the appearance of the sub-cavity containing the element $\{p_2, p_4, p_6\}$. The element is added to the list of elements to terminate the triangulation process. (b) Example of sub-cavity that forms a contour. After a first iteration of the triangulation algorithm, once all facets of the contour are connected to the vertex that corresponds to the highest entry, the contour $\{p_2, p_4, p_6, p_8, p_{10}\}$ is formed. In this case, the triangulation algorithm is called recursively to mesh the new contour. (c) Example of sub-cavity that forms a contour that contains inner points. After a first iteration of the triangulation algorithm, the contour $\{p_0, p_2, p_4, p_6, p_8, p_{10}\}$ is formed that contains the inner points $p_{I,1}$ and $p_{I,2}$. Similarly, the triangulation algorithm is called recursively to mesh the contour with the inner points.

Algorithm 5: Triangulation algorithm used for connecting facets of the contour with inner vertices. The algorithm is coded in *Python.*

```
$A_{ordered}$: ordered connection table row-wise and then column-wise
$F$: set of contour edges
$V$: set of contour vertices and inner vertices
$V_{open}$ : set of vertices open for connections
$V_{locked}$: set of locked vertices
$F_{locked}$: locked facets
Mesh ($F$,$V$,$A_{ordered}$):
    foreach facet $F_i$ (row) in $A_{ordered}$ do
        do
            Connect $F_i$ with $v_j$ (column) to form element $e_{i,j}$
            if every vertex $v$ in $e_{i,j}$ doesn't have the same group id then
                proceed to connect $F_i$ with next $v_j$
            end
            if there is a vertex $v$ in $e_{i,j}$ that belongs to $V_{locked}$ then
                proceed to connect with next $v_j$
            end
            validate $e_{i,j}$
        while $e_{i,j}$ is not validated;
        if $e_{i,j}$ divides cavity to sub-cavities then
            foreach subcavity do
                assign same group id to vertices of sub-cavity
            end
        end
        if a vertex $v$ of $e_{i,j}$ is locked contains another contour facet then
            insert $v$ in $V_{locked}$
            if element $e_{i,j}$ contains another contour facet $F_k$ then
                remove facet $F_k$ ($k_{th}$ row) (tagged as $F_{locked}$) from $A_{ordered}$
            end
        end
    end
    $V_{open}$=V \ $V_{locked}$
    if $V_{open}$ is not empty then
        CheckForSubContours($V_{open}$)
        foreach Sub-contour with edges $F^{'}$ and set of vertices $V^{'}$ do
            Mesh($F^{'}$, $V^{'}$,$A^{'}_{ordered}$)
        end
    end
```

The connection tables A of the training contour datasets are calculated and then used to train NN_3. NN_3 outputs the entries of the connection table A of dimension $d_A = N_C \times (N_C + N_I)$, where N_C is the number of edges of the contour and N_I is the number of inner vertices. NN_3 takes as input the coordinates of the contour P_C and the coordinates of the inner vertices P_I. It first applies convolution and pooling to the coordinates of the contour. The flattened results of pooling along with the coordinates of the inner vertices are then connected with multilayer perceptrons (Fig. 2.13). The network is trained to minimize the loss function $\mathscr{L}(a_{i,j}, \hat{a}_{i,j}) = \sum_{i,j}(a_{i,j} - \hat{a}_{i,j})^2 / N_{d_A}$, where $a_{i,j}$ is the real value of the entry to the connection table A, $\hat{a}_{i,j}$ is the entry that NN_3 predicts, and $N_{d_A} = N_C \cdot (N_C + N_I)$ (Alg. 6).

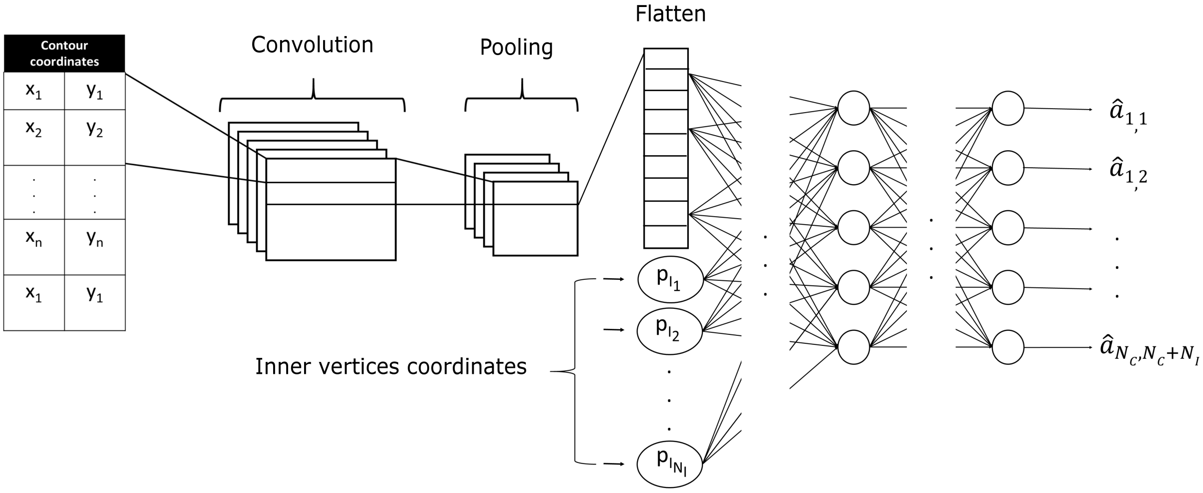


Figure 2.13: NN_3 starts by applying 2-D convolution to the coordinates P_C that are ordered in a circular way. It proceeds by applying a pool function to the convoluted result. The flattened outcome of pooling along with the coordinates of the inner vertices P_I are then connected with multilayer perceptrons. It outputs the entries $a_{i,j}$ of the connection table.

Algorithm 6: Training algorithm of NN_3 for the prediction of connectivity.

```
$P_C$: Contour vertices coordinates
$P_I$: Inner vertices coordinates
$A$: Connection table
$\hat{A}$: Estimated connection table
$N_{train}$: Number of training data population
$N_{d_A}$: Dimension of flattened connection table
Initialize weights $W_K$ of $NN_3$
while required number of iterations is not reached do
    foreach training example in $D = \{(P_C^{(n)}, P_I^{(n)}, A^{(n)})\}_i, i = 1, .., N_{train}$ do
        Compute $\hat{A^{(n)}}$ using current parameters
        Calculate loss function $\mathscr{L}(A^{(n)}, \hat{A}^{(n)}) = ||A^{(n)} - \hat{A}^{(n)}||_2^2 / N_{d_A} = \sum_{i,j}(a_{i,j}^{(n)} - \hat{a}_{i,j}^{(n)})^2 / N_{d_A}$ and $\frac{\partial \mathscr{L}}{\partial W_K}$
        Update $W_K$ using Adam learning rate optimization
    end
end
```

2.6.2 Grid sampling augmentation of the inner vertices

NN_3 takes as input the contour vertices coordinates P_C and the N_I inner vertices coordinates P_I. For each contour, the N_I inner vertices that are inserted during the refinement process of the reference mesher are used along with sets of N_I vertices that are sampled randomly inside the contour. This process of data augmentation is mandatory for the meshing scheme as the vertices provided as input to NN_3 are an approximation of the inner vertices inserted by the reference mesher. Thus, to increase the efficacy of learning, NN_3 must be trained for multiple N_I inner vertices and not only the ones inserted by the reference mesher. The output of NN_3 depends on the order in which the N_I vertices are inserted. The structure of the connection table is column-wise dependent on the order of the input inner vertices of NN_3. As a result, small perturbations of the inner vertices may have a great impact on the mesh connectivity. After considering various ordering methods (angular, coordinates), it is found that none of them are invariant to perturbations of the inner vertices coordinates. Therefore, a sampling process is chosen that selects N_I vertices randomly from the interior of a contour cavity with a target edge length rule (Fig. 2.14). The sampled inner vertices along with the coordinates of the contour are included in the training set of NN_3 without applying any ordering rule to them. This choice improves consistency which facilitates the learning process of NN_3, as the approximation of the entries of the connection table are now based on the actual coordinates of the inner vertices, and do not rely on a specific ordering rule.

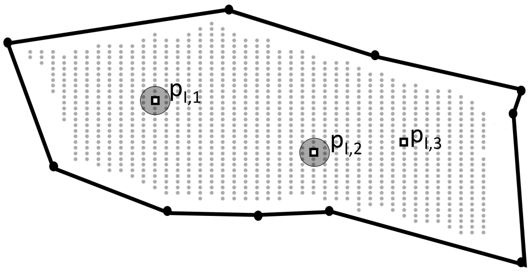
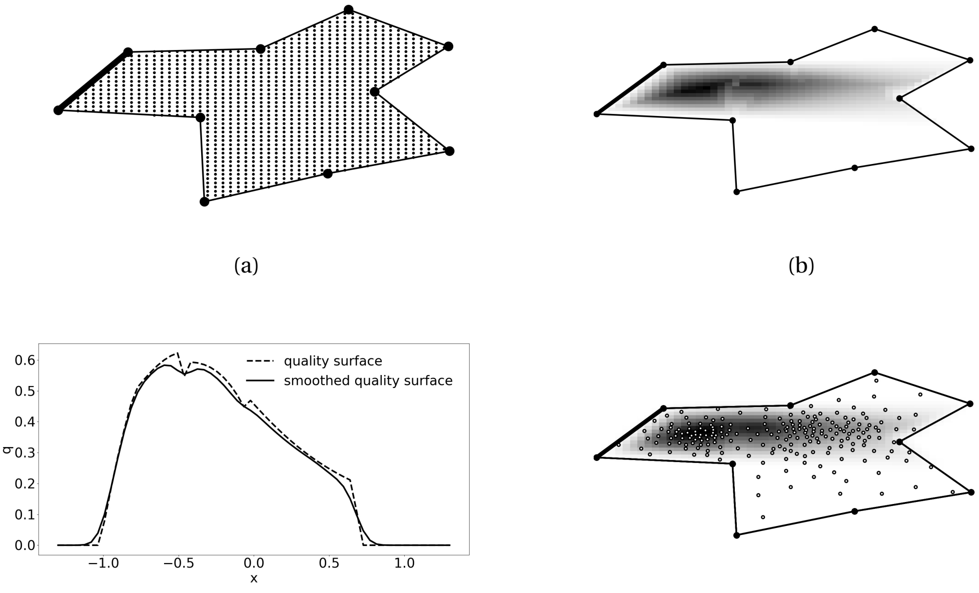


Figure 2.14: Example of sampling inner vertices for the training of NN_3 for $N_I = 3$. From a grid of inner vertices, $p_{I,1}$ is randomly chosen and all the vertices contained at a distance of $0.1 l_s$ from it are excluded from being selected later on. $p_{I,2}$ is then selected, imposing the same exclusion zone, and finally $p_{I,3}$. All the vertices from the grid are at a distance of $0.1 l_s$ from the edges of the contour.

2.6.2.1 Adaptive sampling strategy

Although the process of random sampling facilitates the learning of NN_3, it causes the accumulation of large training populations. In an attempt to reduce the training populations and maintain or improve the accuracy of the connectivity predictions, an alternative method of sampling is also examined. A NN, NN_3^*, is trained that takes as input the coordinates P_C of the contour, the coordinates of an inner vertices $p_{I,j}$, where $j = 1,2..N_I$, and an edge index i, where $i = \{1,2,3...N_C\}$. NN_3^* outputs the entries of the connection table of the i^{th}

row, i.e. the quality values of the mesh that correspond to connections with the i^{th} edge of the contour. Therefore, the predictions of the full connection table are obtained by calling NN_3^* for $i = 1, 2, .., N_C$. On a grid G, for every vertex $g_{i,j} \in G$ that is located inside a contour, the worst quality of the mesh is calculated, given that the vertex is connected with an edge (Fig. 2.15a). In the process, a quality surface S_i is defined, where $i = \{1, 2, ..., N_C\}$, for every edge of a contour (Fig. 2.15b). Each quality surface S_i is smoothed to avoid the appearance of sudden peaks while maintaining the maximum values (Fig. 2.15c). Vertices are sampled from S_i according to the curvature loss criteria, i.e. more vertices are sampled from regions where the rate of change of curvature is high (Fig. 2.15d). The process of sampling of vertices from the smoothed quality surface using the curvature loss criteria is more efficient than that from the un-smoothed surface. This is because peaks with a sudden change in curvature lead to a sampling set of vertices with higher reconstruction loss as compared to that from the smoothed surface. Thus, the smoothing process assists in the training of NN_3^* by providing a set of sampled vertices that provides more representative information on the quality surface within a contour.



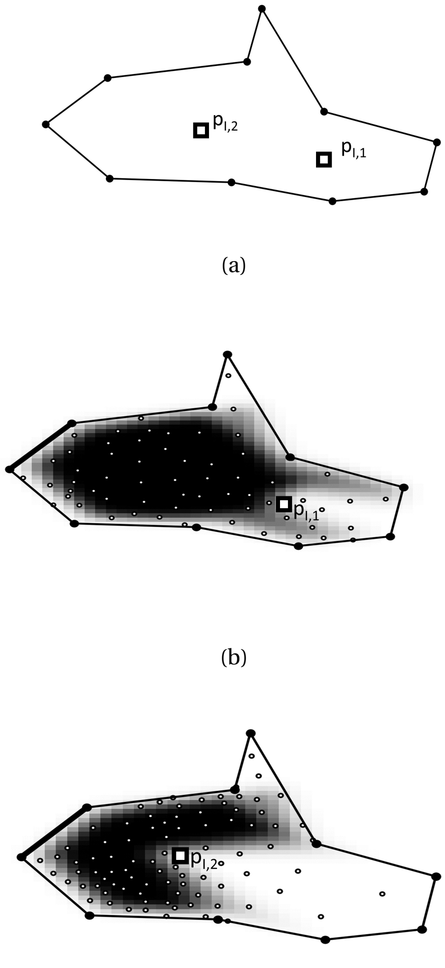
(a) (b)

(c) (d)

Figure 2.15: (a) For the highlighted edge, the worst mesh quality is calculated by connecting this edge with each point of the grid. (b) By doing so, a quality surface is defined for this edge. (c) Curves depicting the quality values q along the original quality surface and the smoothed quality surface for a fixed y-value. By smoothing, sudden peaks are eradicated. (d) Vertices are sampled from the smoothed surface according to curvature criteria; the higher the curvature the more vertices are sampled.

Similarly, for multiple inner vertices, given the position of $N_I - 1$ inner vertices, a quality surface is defined by calculating the minimum quality of each grid vertex from G. Each surface

$S_{n,i}$, where $n = 1,2..,N_I$ and $i = 1,2,3...N_C$, is smoothed and vertices are sampled by using the curvature loss criteria to form sets of inner vertices $V_{n,i}$ (Fig. 2.16). From each set $V_{n,i}$, an inner vertex is sampled. An additional criterion is imposed so that the sampled vertices must be at a distance of at least $0.3l_s$ from one another. The sampled vertices are used to train NN_3^*.



(a)

(b)

(c)

Figure 2.16: (a) Example of adaptive sampling for i_{th} edge (highlighted) of a contour with 10 edges and two inner vertices. (b) Surface $S_{1,i}$ is defined by computing the minimum quality of each grid vertex taking into account $p_{I,1}$ as a second point. By smoothing $S_{1,i}$ and implementing the curvature loss criteria, vertices are sampled from the surface to form the set of inner vertices $V_{1,i}$. (c) In a similar fashion the surface $S_{2,i}$ is defined and vertices from it are sampled to form the set $V_{2,i}$. To train the NN, a pair of vertices with one vertex belonging to $V_{1,i}$ and another belonging to $V_{2,i}$ is sampled. The collected vertices must be at a distance of at least $0.3l_s$ from each other.

3 Results and validation of 2-D simplicial contour Meshing using Neural Networks

3.1 Experimental Conditions

3.1.1 Error metrics

Experiments are conducted in testing populations of random contours. The predictive approximation on the number of vertices using NN_1 is evaluated by calculating the mean absolute error between the predictive value and the real value. For a contour, the predictive approximation of the location of the inner points (NN_2) and the connectivity (NN_3, NN_3^*) are evaluated in terms of quality of the generated mesh using the output values of the respective NN. Two different mesh qualities are calculated, the worst quality $\hat{q}_{worst} = \min_{i\in\{1,..,N_{el}\}} q_{el}^{(i)}$ and the mean quality $\hat{q}_{mean} = \sum_i^{N_{el}} q_{el}^{(i)}/N_{el}$, where N_{el} is the number of total elements and q_{el} is the quality metric as defined in section 2.5.1. The contour is then meshed using the parameters acquired by applying the reference algorithm of CDT followed by refinement to calculate the values q_{worst} and q_{mean}. The relative differences $e_{worst} = (q_{worst} - \hat{q}_{worst})/q_{worst}$ and $e_{mean} = (q_{mean} - \hat{q}_{mean})/q_{mean}$ define the worst and mean triangulation error, respectively, for a contour. e_{worst} and e_{mean} are used as error metrics when approximating with NN_2, NN_3 and NN_3^*. For a population of contours with N_C edges and N_I inner vertices the expected values $E(e_{worst})$ and $E(e_{mean})$ are estimated by calculating the averages $\overline{e}_{worst} = \sum^{n_{N_{C,I}}} e_{worst}/n_{N_{C,I}}$ and $\overline{e}_{mean} = \sum^{n_{N_{C,I}}} e_{mean}/n_{N_{C,I}}$, where $n_{N_{C,I}}$ is the number of meshed contours participating in the test population.

3.1.2 Training dataset populations

Contour training datasets are populations of contours for $N_c = \{4, 6, 8, 10, 12, 14, 16\}$ edges. The population of contours that are generated, are increased at a nearly exponential rate with the number of edges (Fig. 3.1); it is observed that this increase is necessary to retain a level of accuracy from the NNs involved in the meshing scheme so that it provides good quality meshes.

The populations of contours used for the training of NN_1 (Fig. 3.1) are divided into groups

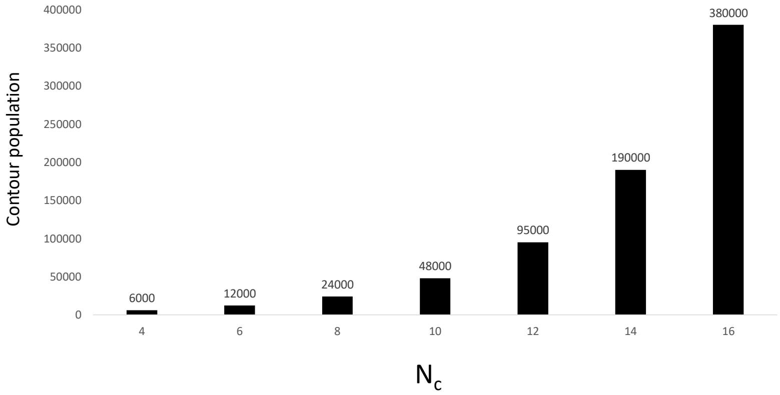


Figure 3.1: Histogram of contour populations with N_C edges that are used for training. The population of 6000 contours with 4 edges is found to be an adequate training set for acquiring satisfactory accuracy from the NNs involved in the meshing procedure. To retain or acquire a level accuracy needed for good quality meshing, for contours with 6 and 8 edges a training population of approximately 12000 and 24000 contours, respectively, is required; this leads to the choice of generating contour populations used for training that increase exponentially with the number of edges.

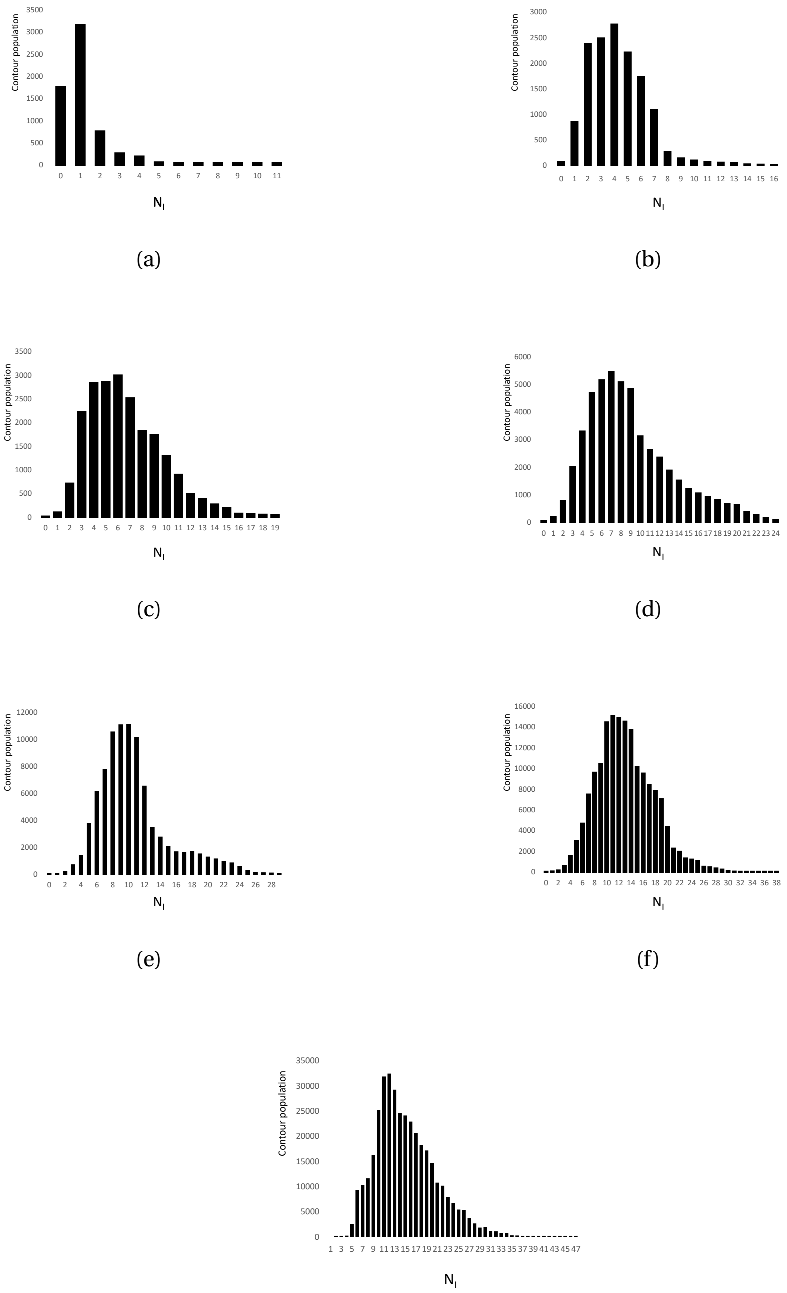
with respect to the number of inner vertices inserted by the reference mesher during the refinement process for target edge lengths ranging from 0.2 to 1 (Fig. 3.2). Each of these groups is then used to train NN_2 and NN_3.
The training data were generated in a machine with 64 GB memory and 2 CPUs Intel© Xeon© E5-2660v2 with 2.2 GHz and 10 cores and a machine with 128GB memory with the same 2 CPUs.

3.1.3 NN_i hyperparameters

NN_1 is trained for 3000 epochs with a learning rate of $\eta = 10^{-4}$ and a weight decay of $\lambda = 10^{-1}$ with a batch size of $n_{batch} = 512$. There are 3 layers with batch normalization and the ReLU activation function for the first 2 layers with $4 \cdot N_C$ hidden nodes.

For the training of NN_2, the grid G used for the approximation of the inner vertices, is divided into N_p patches $G_k, k = \{1, 2..N_p\}$. NN_2 is trained for 5000 epochs with a learning rate of $\eta = 10^{-4}$ and a weight decay of $\lambda = 10^{-2}$ with a batch size of $n_{batch} = 512$. There are 3 layers with batch normalization and the ReLU activation function for the first 2 layers. The first two layers contain $2 \cdot N_C + N_{G_k}$ hidden nodes and the output layer contains N_{G_k} nodes, where N_{G_k} is the number of grid points contained in the patch G_k.

The populations of contours used for training NN_3 are the same as the ones used for the



(a)

(b)

(c)

(d)

(e)

(f)

(g)

Figure 3.2: (a)-(g) In alphabetical order, the histogram of the contour populations for $N_C = (4, 6, 8, 10, 12, 14, 16)$ number of edges (Fig. 3.1) divided into groups according to the number of vertices N_I that are inserted by the reference mesher. Each of these groups is used to train NN_2 and NN_3.

training of NN_2 (section (3.1.2)) (Fig. 3.2). NN_3 is trained for 5000 epochs with a learning rate of $\eta = 10^{-4}$ and a weight decay of $\lambda = 10^{-1}$ with a batch size of $n_{batch} = 512$. The convoluted layer contains N_C number of filters. The convolution is done using a stride of 2 and filter size 2×2. Max pooling is then applied with stride 2 with 2×1 filters to the convoluted result. There are 3 fully connected layers with batch normalization and the tanh activation function for the first 2 layers. The number of hidden nodes for the first two layers is $2 \cdot N_C \cdot (N_C + N_I)$. The output layer contains $N_C \cdot (N_C + N_I)$ nodes which is the flattened dimension of the connection table.

3.2 Results

3.2.1 Predictions of the number of inner vertices

The prediction of the number of inner vertices from NN_1 is evaluated on groups of random contours with N_C edges. The confidence level and confidence interval of each test population is 95% and 5%, respectively, of each training population as presented in section (3.1.2) (Fig. 3.1). The random contours are meshed by applying CDT with refinement for target edge lengths ranging from 0.2 to 1. The mean absolute error $\overline{e}$ is examined. The mean absolute error is defined as $\overline{e} = \overline{|N_I - \hat{N}_I|} = \sum_{i=1}^{n_{N_C}} (N_I^{(i)} - \hat{N}_I^{(i)}) / n_{N_C}$, where $\hat{N}_I = (\hat{N}_I^{(1)}, ..., \hat{N}_I^{(n_{N_C})})$ are the approximations of NN_1 and n_{N_C} is the number of contours with N_C edges in the test population. By examining $\overline{e}$ it is observed that the error increases with the number of edges (Fig. 3.3). For example, for a target edge length of $l_s = 0.2$, the mean absolute error $\overline{e}$ for the contour with 16 edges is approximately 2.3 times higher than that for the contour with 4 edges. This is attributed to the fact that there is a much larger variation in the number of vertices N_I being inserted by the reference mesher during refinement for the contour population of 16 edges than the contour population with 4 edges. The standard deviation of N_I points being inserted is $std_{16} = 5.2$ and $std_4 = 1.4$ for the contour populations with 16 and 4 edges, respectively.

Moreover, in every population of contours with N_C edges, the error is higher for smaller target edge lengths l_s. As explained before, this increase in the error is due to the larger variation of N_I points being inserted by the refence mesher during refinement inside a contour population with N_C edge length, as the target edge length gets smaller; e.g. for the contour population of 14 edges the mean absolute error $\overline{e}$ for $l_s = 0.2$ and $l_s = 1$ decreases from 1.3 to 0.2 and the standard deviation of N_I for these target edge lengths decreases by 12%. Despite the increase in the mean absolute error $\overline{e}$ with increase in the number of contour edges and decrease in the target edge length, the predictions of NN_1 are considered accurate enough to be used for meshing purposes (Fig. 3.4); e.g. for a random contour of 16 edges the number of vertices of the graded reference mesh and the number of vertices predicted by NN_1 differ by one for target edge lengths $l_s = \{0.2, 0.3, 0.4\}$ and is the same as the target edge length increased (Fig. 3.4b).

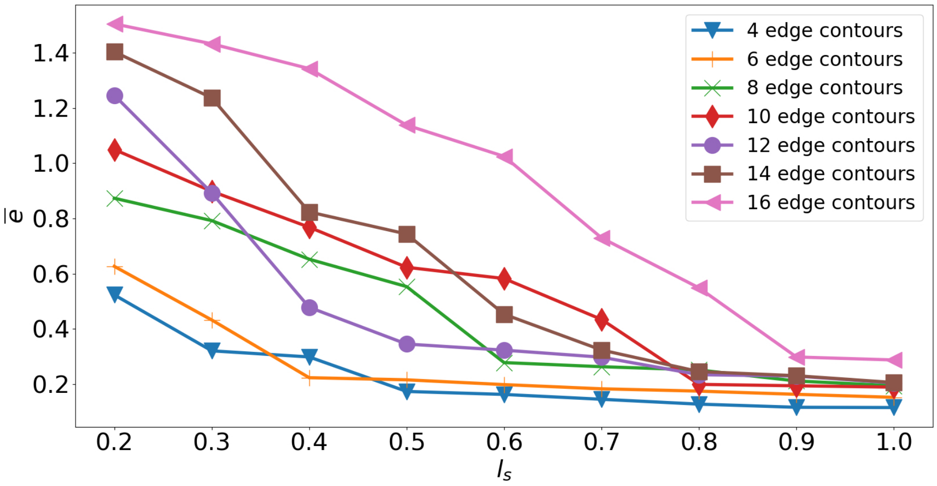
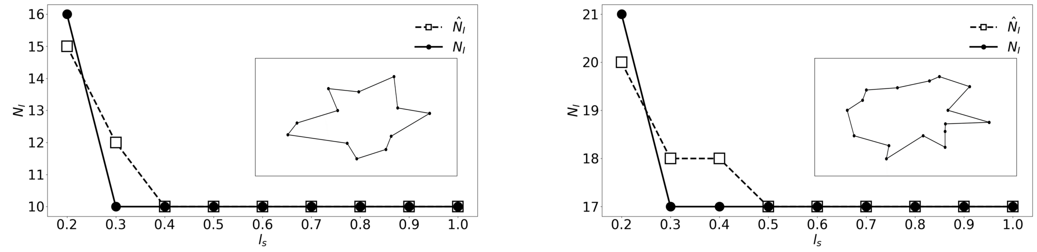


Figure 3.3: The mean absolute error $\overline{e}$ as a function of target edge lengths l_s ranging from 0.2 to 1. The mean absolute error is defined as $\overline{e} = \overline{\left|N_I - \hat{N}_I\right|} = \sum_{i=1}^{n}(N_I^{(i)} - \hat{N}_I^{(i)})/n$, where $\hat{N}_I = (\hat{N}_I^{(1)}, \ldots, \hat{N}_I^{(n_{N_C})})$ are the number of vertices predicted by the NN, $N_I = (N_I^{(1)}, \ldots, N_I^{(n_{N_C})})$ are the number of vertices inserted from the reference mesher during refinement and n_{N_C} the number of contours with N_C edges in the test population. The mean absolute error $\overline{e}$ increases with increase in the number of contour edges N_C. For example, for a target edge length $l_s = 0.2$, $\overline{e}$ for the population of contours with 16 edges is approximately 2.3 times higher than the mean error for the population of contours with 4 edges due to the larger variation of number of inner vertices N_I that the reference mesher inserts for the populations of 16 edges. $\overline{e}$ also increases with the decrease of the target edge length l_s which is also due to the fact that the variation on the number of inner vertices N_I inserted by the reference mesher during refinement is larger for smaller target edge lengths. For instance, for the population of contours with 14 edges and the target edge lengths $l_s = 0.2$ and $l_s = 1$, $\overline{e}$ decreases from 1.3 for 0.2 respectively, while the standard deviation of inner vertices N_I for these target edge lengths decreases by 12%.



(a)

(b)

Figure 3.4: The real number of inner vertices N_I and the predicted number of inner vertices $\hat{N}_I$ as a function of the target edge length for a random contour with 12 edges (a) and a random contour of with 16 edges (b). For the contour with 12 edges the N_I and $\hat{N}_I$ differ by one point for $l_s = 0.2$, two points for $l_s = 0.3$ and are the same for the rest of the target edge lengths. For the contour with 16 edges, N_I and $\hat{N}_I$ differ by one point for the target edge lengths $l_s = \{0.2, 0.3, 0.4\}$ and are same for the rest of the values l_s. It can be concluded that NN_1 is appropriate to use for meshing purposes.

3.2.2 Prediction of the inner vertices positions

The grid used for the approximation of the inner vertices P_I has a bounding box $[-1.2, 1.2] \times [-1.2, 1.2]$ which includes all the vertices P_C of the contours. The resolution of the grid affects the accuracy of the predictions. To find an appropriate resolution, the accuracy of NN_2 for the prediction of one inner vertex $p_{I,1}$ on $n = 100$ random contours is examined. The mean squared error of the euclidean distance $\overline{e}_{dist}$ is examined for grid resolutions of of 10×10, 20×20, and 40×40. The mean squared error is defined as $\overline{e}_{dist} = \overline{||p_{I,1} - \hat{p}_{I,1}||} = \sum_i^n (p_{I,1}^{(i)} - \hat{p}_{I,1}^{(i)})^2/n$, where $\hat{p}_{I,1} = (\hat{p}_{I,1}^{(1)}, ..., \hat{p}_{I,1}^{(n)})$ are the approximations of the inner vertex for n contours with N_C edges. $\overline{e}_{dist}$ is calculated for the aforementioned grid resolutions . By examining the convergence rate of $\overline{e}_{dist}$, it is concluded that for the case studies a cell size of 20×20 is adequate to get accurate results (Fig. 3.5). This resolution also ensures that the distance between successive grid points is 33% smaller than the smallest target edge length $l_s = 0.2$. Once a grid point with minimum score is selected, all the grid points contained at a distance of $0.2l_s$ participate in the interpolation process. The scores of all the grid points that are contained within a distance of $0.1l_s$ from the edges of the contour are penalized. Finally, all the grid points contained closer than a distance of $0.1l_s$ from a chosen grid point with minimum score are excluded as possible candidates for approximation of the next inner vertex. The grid is divided into 100 patches. Each of the patches contain 4 grid points.

To measure the triangulation error caused by the approximations of NN_2 the averages errors $\overline{e}_{worst}$ and $\overline{e}_{mean}$ are measured between the meshes generated using the approximated inner vertices $\hat{P}_{I,i}$, where $i = \{1, 2..., N_I\}$, and the meshes generated using the inner vertices $P_{I,i}$ that are inserted from the reference mesher during refinement (Fig. 3.6). All the tests are conducted on random contours with a sample size of 95% confidence level and 5% confidence interval of each training group presented in section (3.1.2) (Fig. 3.2).

The triangulation error in the prediction of inner vertices is due to two factors: (1) The resulting

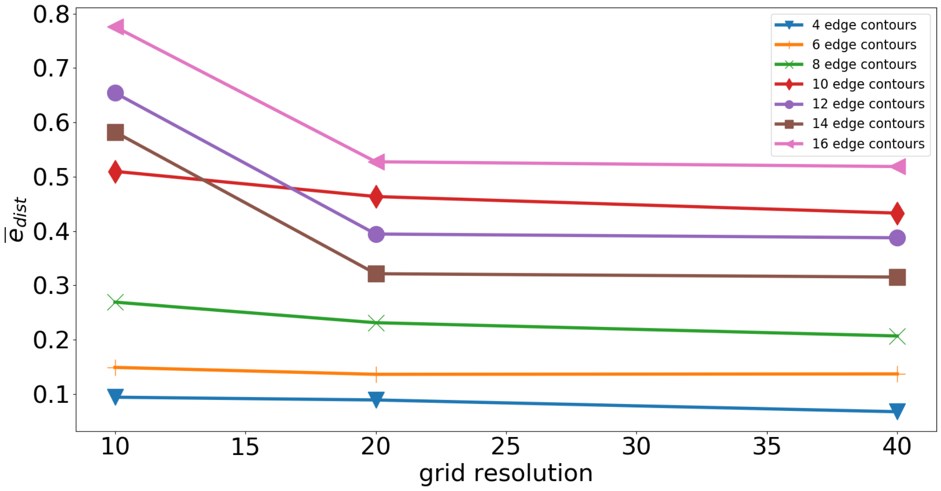
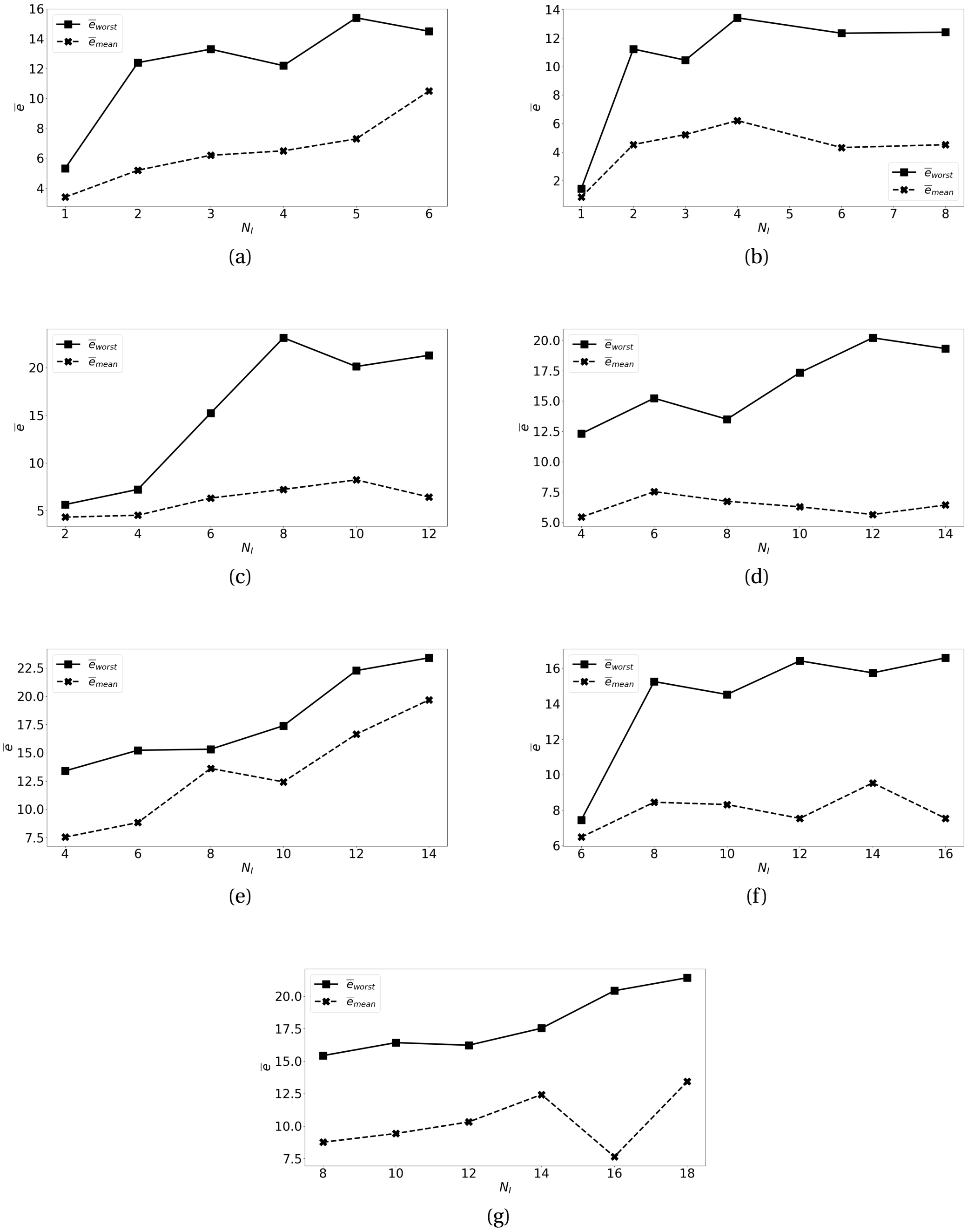


Figure 3.5: The mean squared error of euclidean distance $\overline{e}_{dist}$ as a function of grid resolutions of 10 × 10, 20 × 20, and 40 × 40. The mean square error is defined as $\overline{e}_{dist} = \overline{||p_{I,1} - \hat{p}_{I,1}||} = \sum_i^n (p_{I,1}^{(i)} - \hat{p}_{I,1}^{(i)})^2/n$, where $p_{I,1} = (p_{I,1}^{(1)}, ..., p_{I,1}^{(n)})$ are the real locations of the inner vertex inserted by the reference mesher during refinement and $\hat{p}_{I,1} = (\hat{p}_{I,1}^{(1)}, ..., \hat{p}_{I,1}^{(n)})$ are the predictions of the inner vertex for $n = 100$ contours with N_C edges. For every contour population, the error reaches convergence by using a grid with resolution 20 × 20.

mesh with the predicted inner vertices may have the same connectivities as the mesh with the inner vertices inserted by the reference mesher during refinement. In this case, the approximated inner vertices form triangular elements that are of worse quality due to their displacement (Fig. 3.7). (2) Applying CDT with the approximated inner vertices may result in a mesh with different connectivities altogether, where worse quality elements appear. The average error of $\overline{e}_{worst}$ increases with the number of inner vertices N_I, reaching a maximum of 23.41% for the case of contours with 12 edges and 14 inner vertices (Fig. 3.6e). This increase demonstrates how the complexity of a mesh affects the approximations of NN_2; the displacement error accumulated by the prediction of the inner vertices increases with the number of inner vertices, which in turn causes the increase in $\overline{e}_{worst}$. This effect is mostly noticeable in the case of contours with 6 edges with one inner vertex and 2 inner vertices (Fig. 3.6b). Merely by the addition of one inner vertex, $\overline{e}_{worst}$ is increased from 1.6% to 10.5%. A few mild fluctuations in $\overline{e}_{worst}$ within the populations of N_C edges are explained by the choice of random contours that belong to the test dataset (e.g. for the case of contours with 14 edges with 3 and 4 inner vertices) (Fig. 3.6a) or because there is an increase in the training population as the number of inner vertices is increased (e.g. for the case of contours with 14 edges containing 8 and 10 inner vertices) (Fig. 3.6f). In addition, it is observed that $\overline{e}_{mean}$, which has a maximum of 19.7% in the case of a contour with 12 edges and 14 inner vertices (Fig. 3.6e), does not necessarily follow the pattern of $\overline{e}_{worst}$ (Fig. 3.6d). It is expected for $\overline{e}_{mean}$ to be lower than $\overline{e}_{worst}$, as the mean quality q_{mean} takes into account the qualities of all the elements in the mesh and not just that of the worst element.



(a) (b) (c) (d) (e) (f) (g)

Figure 3.6: (a)-(g): In alphabetical order, the average errors $\overline{e}_{worst} = \sum^{n_{N_{C,I}}} e_{worst}/n_{N_{C,I}}$ (%) and $\overline{e}_{mean} = \sum^{n_{N_{C,I}}} e_{mean}/n_{N_{C,I}}$ (%) for a number of $n_{N_{C,I}}$ random contours with $N_C = \{4, 6, 8, 10, 12, 14, 16\}$ edges as a function of inner vertices N_I. The range of inner points covers at least 68% of each contour population with N_C edges. Maximum $\overline{e}_{worst}$ of 23.41% occurs for the case of contours with 12 edges and 14 inner vertices (e). In most cases, both $\overline{e}_{worst}$ and $\overline{e}_{mean}$ tend to increase with the increase in the number of inner vertices; the displacement error from the predicted vertices of NN_2 increases with the number of vertices N_I , which, in turn, increases the triangulation errors $\overline{e}_{worst}$ and $\overline{e}_{mean}$.

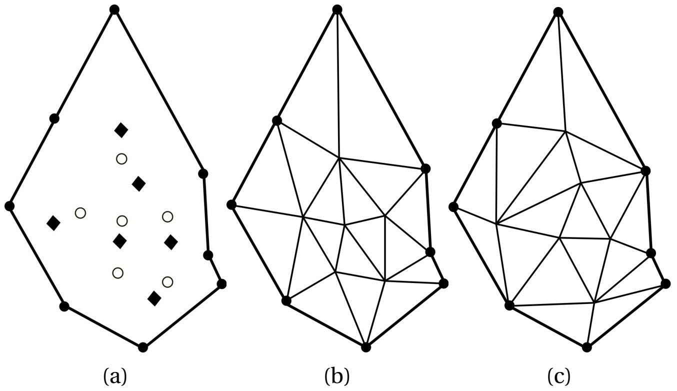


Figure 3.7: (a) A contour with 8 edges with inner vertices inserted by the reference mesher during refinement (circular points) and inner vertices approximated by the scores of NN_2 (diamond points). (b) The graded reference mesh with its inner vertices. (c) The resulting mesh with approximated vertices. In this case, the triangulation errors account for $e_{worst} = 22.3\%$ and $e_{mean} = 9.8\%$.

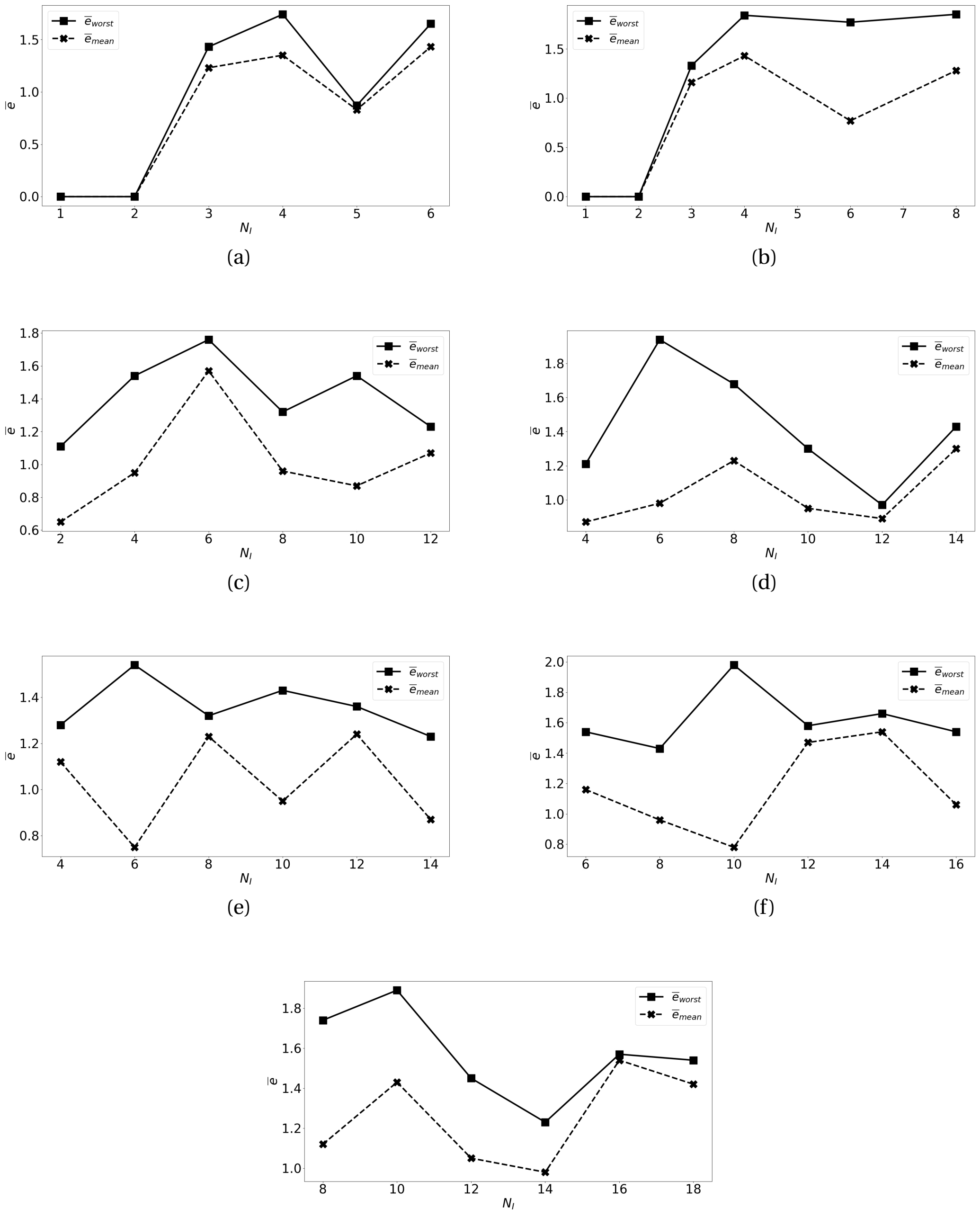
3.2.3 Prediction of the connectivity

To estimate the triangulation error caused by the triangulation algorithm, the average errors, $\overline{e}_{worst}$ and $\overline{e}_{mean}$, are measured between the meshes generated with the calculated connection table A and the graded reference meshes on the same random contours as those used in section 3.2.2. A maximum triangulation error $\overline{e}_{worst} = 1.9\%$ is reached in the case of contours with 14 edges and 10 inner vertices (Fig. 3.8f) and a maximum of $\overline{e}_{mean} = 1.5\%$ for contours with 16 edges and 16 inner vertices (Fig. 3.8g). In some cases, there is no triangulation error (Fig. 3.8a, Fig.3.8b). The fluctuation in the errors may be caused by the random choice of contours participating in the test dataset. The level of triangulation error in $\overline{e}_{worst}$ and $\overline{e}_{mean}$ demonstrates that the algorithm does not account for a significant error propagation in the connection scheme.

The number of sampled vertices chosen for a contour increases according to the number of vertices N_I that are intended to be inside the cavity. 50 N_I groups of sample vertices are chosen for $2 \leq N_I \leq 4$, 100 for $5 \leq N_I \leq 8$, and 200 for $N_I \geq 9$.

The triangulation error caused by the approximations of NN_3 is tested for the random contours used in section 3.2.2 which contained the N_I inner vertices that are inserted by reference mesher during refinement. The average triangulation errors, $\overline{e}_{worst}$ and $\overline{e}_{mean}$, are calculated between the meshes that are generated with entries of the connection table that are predicted using NN_3 and the graded reference meshes. A maximum triangulation error $\overline{e}_{worst} = 24.04\%$ is reached for the contour with 8 edges and 12 inner vertices. It is observed that there are more fluctuations as compared to the triangulation errors caused by the prediction of NN_2. The following observations are made upon examining the average triangulation error $\overline{e}_{worst}$:

- For contours with N_C edges and N_I inner vertices, the accuracy of the connectivity depends on the number of sampled vertices N_I used to train NN_3. This is quite evident



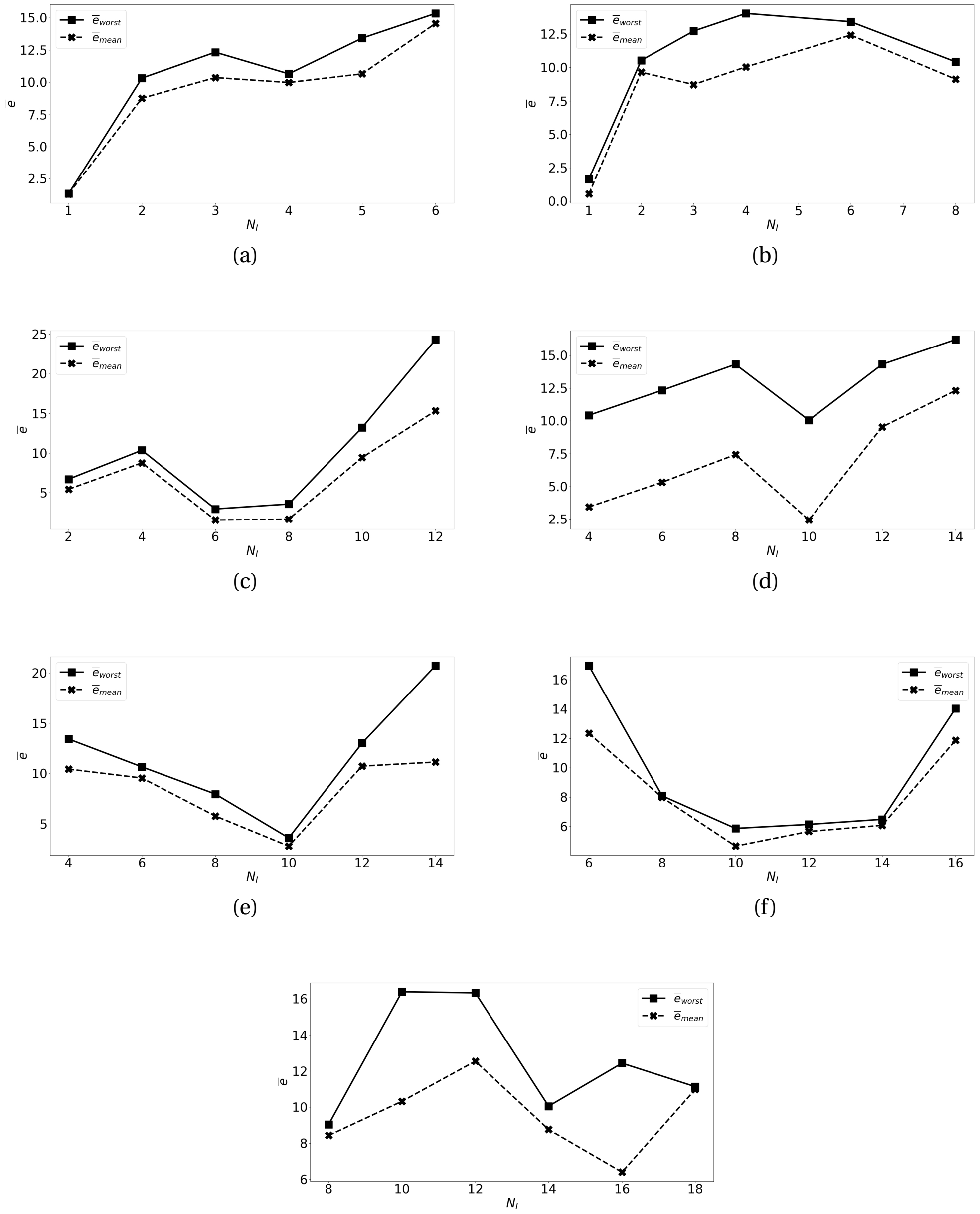
(a) (b)

(c) (d)

(e) (f)

(g)

Figure 3.8: (a)-(g): Validation of the developed triangulation algorithm. The average triangulation errors, $\overline{e}_{worst}$ and $\overline{e}_{mean}$, between meshes that are generated by using the triangulation algorithm with calculated connection table and the graded reference meshes for random contours with $N_C = \{4, 6, 8, 10, 12, 14, 16\}$ edges as a function of number of inner vertices N_I. The levels of triangulation errors indicate that the triangulation algorithm causes little to no significant error propagation in the connection scheme.



(a)

(b)

(c)

(d)

(e)

(f)

(g)

Figure 3.9: (a)-(g): Validation of the connectivity network NN_3. The average triangulation errors, $\overline{e}_{worst}$ and $\overline{e}_{mean}$, between contours meshed by using the triangulation algorithm (predicted connection table by NN_3) and the graded reference meshes for random contours with $N_C = \{4, 6, 8, 10, 12, 14, 16\}$ edges as a function of the number of inner vertices N_I. The inner vertices are those inserted in the cavity by the reference mesher during refinement. The accuracy of NN_3 is dependent on the number of N_I sampled inner vertices of a contour during the data augmentation process. For example in the case of contours with 6 edges for $N_I = 6$ (b) the training contour population is for 3% lower than that for $N_I = 4$ but there is reduction of 23% of $\overline{e}_{worst}$; this is due to the fact that the training of NN_3 for $N_I = 6$ relies on sampling 100 groups of six inner vertices for each contour, whereas for $N_I = 4$, 50 groups of four inner vertices are sampled.

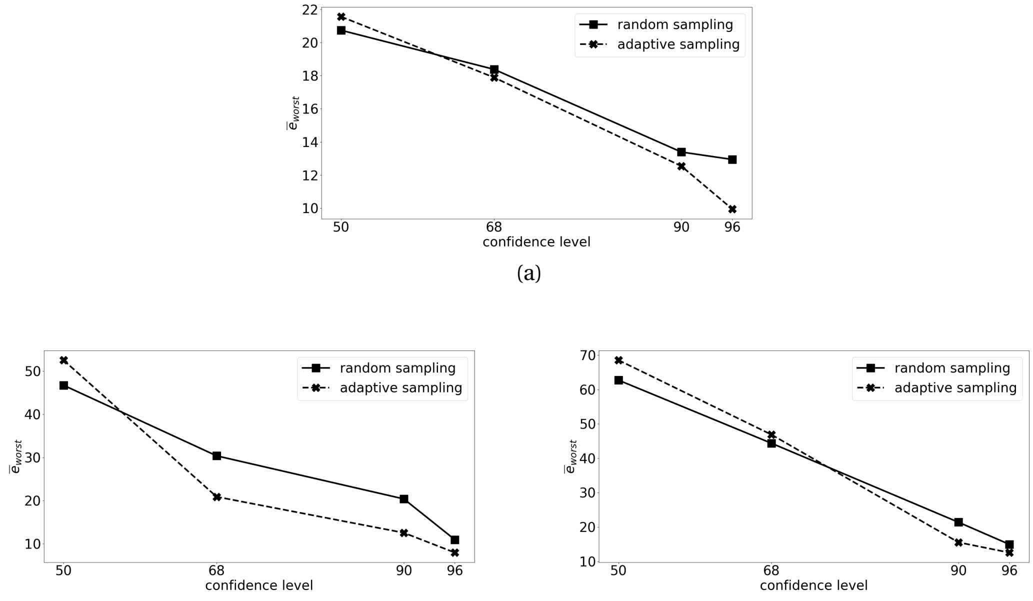
in the case of contours with 6 edges for $N_I = \{4,6\}$ (Fig. 3.9b). Although the training contour population for $N_I = 6$ inner vertices is 37% lower than that for $N_I = 4$, there is a 23% reduction in $\overline{e}_{worst}$ as the number of sampled inner vertices for $N_I = 6$ is twice as big as for $N_I = 4$. The same is observed in the case of contours with 10 edges for $N_I = \{8,10\}$ (Fig. 3.9c). Even though the complexity of the triangulations is increased (i.e. the number of inner vertices) and the training contour population for $N_I = 10$ is 38% lower than that for $N_I = 8$, $\overline{e}_{worst}$ decreases by 29%.

- For a fixed number of N_I sampled inner vertices, $\overline{e}_{worst}$ increases with increase in the number of vertices or decrease in the training data. For example, for the contour with 12 edges with $N_I = \{10,12\}$ (Fig. 3.9e), $\overline{e}_{worst}$ increases from 3.6% to 13% whereas the population of the training contours decreases by 41%. However, for the same contour population for $N_I = \{6,8\}$, even though the complexity of triangulation is increased, $\overline{e}_{worst}$ decreases from 10.7% to 8% whereas the training contour population is increased by 42%. The accuracy of NN_3 appears to be more dependent than NN_2 on the size of the training contour populations (Fig. 3.2).

3.2.4 Efficiency of adaptive sampling

To examine the efficacy of the adaptive sampling, the predictions of NN_3^* are studied for the case of contours with 10 edges with one inner vertex for a training population of 2000 contours. On a grid G of resolution 50×50, the worst quality of the mesh is calculated, given that the vertex is connected with an edge (Fig. 2.15a). First, NN_3^* is trained with inner vertices that are sampled randomly from the grid for different sample sizes of the whole population of inner vertices (Table 3.1). Next, NN_3^* is trained by providing every i^{th} edge index with vertices that are sampled adaptively from S_i by using the curvature criteria. For each edge, 45 inner vertices are sampled adaptively i.e. the whole population of inner vertices for a contour is 450. NN_3^* is trained for different sample sizes of this population. The accuracy of NN_3^* is tested on random contours that accounted for a 95% confidence level and 5% for each training population. Using adaptive sampling, the accuracy is improved by 27% while requiring 35% less samples with respect to the use of random sampling. (Fig. 3.10a).

Next, adaptive sampling is applied for the case of contours with 10 edges with $N_I = \{2,4\}$ inner vertices and a training population of 2000 to determine the accuracy of NN_3^*. Given the position of $N_I - 1$ inner vertices, a quality surface is defined by calculating the minimum quality of each grid vertex from G. For each edge 45 vertices are sampled from each surface $S_{n,i}$. Thus, for each contour, the whole population is $45 \times N_I \times 10$ inner vertices. The accuracy of NN_3^* is tested on random contours that accounted for 95% confidence level and 5% for each training population. Once again, using adaptive sampling leads to 18% and 27% better accuracy for $N_I = 2$ and $N_I = 4$ respectively, with populations that are 22% and 17% lower than the ones of random sampling (Fig. 3.10b, Fig.3.10c).



(a)

(b)

(c)

Figure 3.10: Average worst triangulation error $\overline{e}_{worst}$ when NN_3^* is trained for different confidence levels of inner vertices populations (Table 3.1) with random and adaptive sampling. (a) For the prediction of the connectivity with one inner vertex at 96% confidence level, the sample population of inner vertices with adaptive sampling is 35% lower than that with random sampling. Even though NN_3^* is trained with lesser number of samples, 27% better accuracy is achieved. (b) For the prediction of the connectivity with two inner vertices at 90% confidence level, 18% better accuracy is achieved for a 22% lower sample population by applying adaptive sampling. (c) The accuracy for prediction of the connectivity with four inner vertices at 90% confidence level is 27% higher for a 17% lower sample population with adaptive sampling as compared to random sampling.

N_I	Sampling method	Mean population of inner vertices	Confidence level	Sample size
1	Random sampling	689	50	206
			68	332
			90	495
			96	594
	Adaptive sampling	450	50	178
			68	265
			90	359
			96	382
2	Random sampling	1378	50	242
			68	437
			90	772
			96	889
	Adaptive sampling	2 × 450	50	252
			68	374
			90	595
			96	662
4	Random sampling	2756	50	265
			68	519
			90	1071
			96	1312
	Adaptive sampling	4 × 450	50	252
			68	472
			90	888
			96	1047

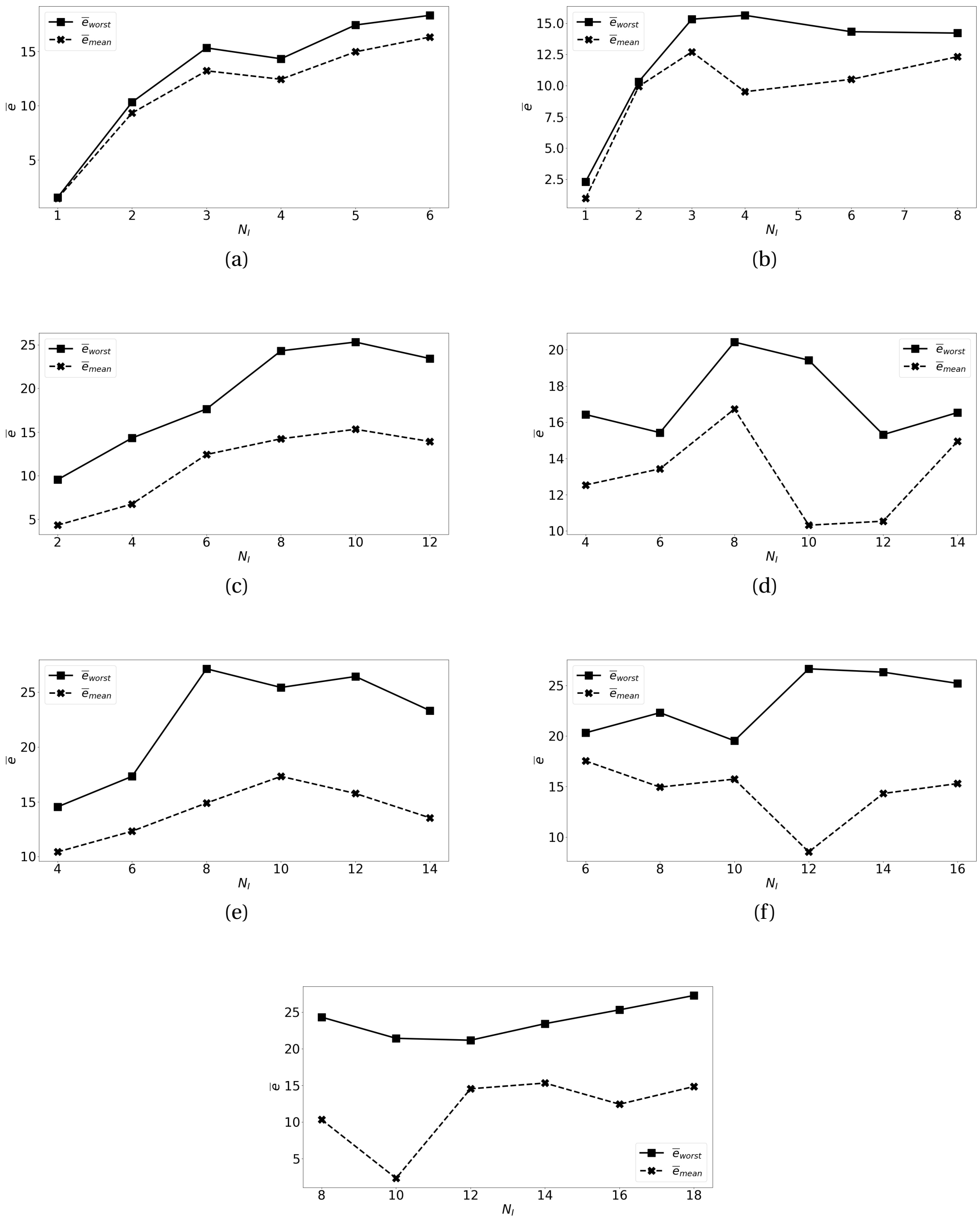
Table 3.1: Sampling sizes for the mean populations of the contours inner vertices with random and adaptive sampling for different confidence levels.

3.2.5 Overall prediction of the meshing scheme

The meshing scheme is tested for populations of contours with $N_C = (4, 6, 10, 12, 14, 16)$ edges for target edge lengths varying from 0.2 to 1.0 (Fig. 3.11) with a sample size of 95% confidence level and 5% confidence interval of each training group (Fig. 3.2). The number of inner vertices N_I, their coordinates, and entries of the connection table are predicted using NN_1, NN_2, and NN_3, respectively. By using all of the NNs involved in the meshing scheme, an increase in the triangulation error compared to our previous tests for every contour population is observed, e.g. for the case of contours with 10 edges and 8 inner vertices (Fig. 3.11g), the average worst triangulation error is $\overline{e}_{worst} = 19.4\%$ whereas the average worst triangulation errors caused by NN_2 and NN_3 are 17.3% and 10%, respectively. The increase in the error is not only due to the approximation of the connectivity using NN_3 but also because of the triangulation error caused by using NN_2 for the approximation of inner vertices. A maximum triangulation error $\overline{e}_{worst} = 27.3\%$ is obtained for the contour population of 16 edges with 18 inner vertices (Fig. 3.11g).

3.3 Conclusions

A novel machine learning scheme was presented for meshing 2D simplicial contours of given target edge length. The meshing scheme uses three NNs that predicted the number of inner vertices, their location, and the entries of a connection table. Based on the entries of the connection table, a triangulation algorithm is applied to mesh the contour. The proposed meshing scheme generates topologically valid meshes with no element intersections (manifold



(a) (b) (c) (d) (e) (f) (g)

Figure 3.11: (a)-(g): The average triangulation errors, $\overline{e}_{worst}$ and $\overline{e}_{mean}$, between meshes generated with the meshing scheme and the graded reference meshes for random contours with $N_C = \{4, 6, 8, 10, 12, 14, 16\}$ edges as a function of number of inner vertices N_I. The inner vertices are those predicted by NN_2. In all cases, there is an increase of $\overline{e}_{worst}$ and $\overline{e}_{mean}$ compared to the previous tests where NN_2 and NN_3 are used separately. A maximum $\overline{e}_{worst}$ of 27.3% occurs for the case of contours with 16 edges and 18 inner vertices.

mesh) and can be trained using any classic mesh generator given that no additional vertices are inserted into the edges of the contour. The meshing scheme avoids the incremental creation of a mesh because the location of the inserted inner vertices is predicted and known prior to the connection phase. Heavy post improvements are avoided because inner vertices location and connectivities are trained such as to maximize mesh quality. The algorithm is compact and easily transportable.

The accuracy of every NN involved in the meshing scheme and that of the overall meshing scheme is determined based on a reference mesher that applied CDT followed by a refinement process. It is demonstrated that the maximum minimum triangulation error of the overall meshing scheme is 27.3% for the studied contour population. To give a clearer insight into the accuracy of the meshing scheme, given the quality metric q_{el} used (Section 2.5.1) for an ideal mesh composed of regular triangular elements with 60° angles, the aforementioned level of error corresponds to variation in angles between 28° and 106° in the worst case. For the prediction of the number of vertices with NN_1, a maximum mean absolute error $\overline{e} = 1.5$ occurs in the case of contours with $N_C = 16$ edges and a target edge length $l_s = 0.2$; the number of inner vertices for the meshes generated from the scheme and the meshes from the reference mesher differs at most by two. For the prediction of the location of the inner vertices, a maximum minimum triangulation error 23.41% occurs in the case of contours with $N_C = 12$ edges and $N_I = 14$ inner vertices with the use of NN_2. For the prediction of the connectivity of the mesh using NN_3, a maximum minimum triangulation error 24.04% is observed for contours with $N_C = 8$ edges and $N_I = 12$ inner vertices. These levels of error confirm the reasonably good quality of meshes generated by the scheme. Growth in mesh complexity (i.e. increase in the number of contour edges and number of inner vertices) requires an exponential increase of training data with the number of N_C edges to reach a reasonably good level in quality. Furthermore, the pairs of N_I inner vertices per contour used for the training of connectivity predictions, increases with the number N_I, reaching a maximum of 200 groups of N_I inner vertices for $N_I > 9$; although these pairs are chosen with a target edge length criterion, their selection is random. For the prediction of the connectivity, adaptive sampling allows to increase the accuracy while using significantly less data. To further optimize the accuracy of the overall scheme and reduce the accumulation of large training sets, it is worth investigating more strategic approaches for the selection of contour populations and pairs of inner vertices used for the training procedure. The proposed procedure might still require a large dataset for contour with a high number of contour vertices. However, up to a certain amount of vertices on a contour, it is meaningful to mesh regions of a domain separately, as the meshes generated in two mesh regions located far apart are expected to be weakly correlated.

Performance tests indicate that the proposed meshing scheme is approximately four times slower than the reference mesher. The meshing scheme is coded in *Python* while the reference mesher is written in C++; assuming a speed factor of 5 to 20 between *Python* and *C++*, and that the current implementation of the algorithm is not optimized for performance, the aforementioned difference in speed validates that the scheme attains reasonably good performance. There is also a large potential to increase the speed of the meshing algorithm as it is transportable in terms of code and memory to acceleration platforms such as GPU and FPGA architectures.

4 2-D Local Mesh Improvement using Neural Networks

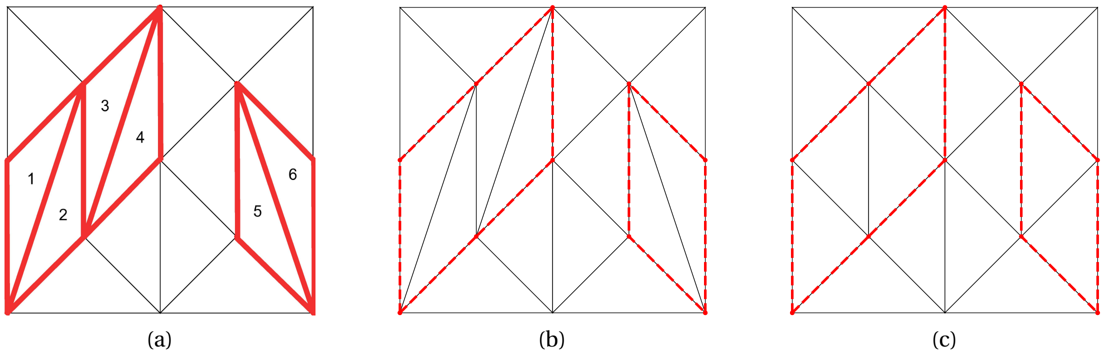
Based on the connectivity NN, NN_3, and a set of new NNs, NN_S and NN_S^*, a set of geometrical operations are developed to improve the quality of a mesh. First, the mesh is partitioned into local mesh configurations in accordance to the applied operation. The local mesh configurations include either elements that are below a quality threshold or edges that are too small or too big according to edge length thresholds. Next, the contours of local mesh configurations are extracted and the operation is applied to them. The operations include:

- **Reconnection**: Using the connectivity network NN_3, the contour of a local mesh configuration that includes low quality elements and no inner vertices is retriangulated.
- **Vertex repositioning**: Using NN_S, an inner vertex of the mesh is repositioned by predicting a new optimal position such that the quality of the adjoined elements improves. The operation is applied to contours of local mesh configuration that include the element that are connected to the vertex.
- **Surface control**: Similar to the vertex repositioning operator, a new position for a boundary or interface vertex is predicted that improves the quality of adjoined elements using NN_S^*. The new position of the vertex is constrained to belong to the boundary or interface that it is part of.
- **Size control**: The size control operation regulates the length of edges that are too small or too big. First, it inserts a vertex in the middle of the target edge. Next, the contour with the inserted inner vertices of the local mesh configuration that include either the short or long edges is retriangulated using NN_3.

4.1 Local Mesh Improvement Operations

4.1.1 Reconnection

The reconnection operation starts with partitioning the mesh according to an element quality threshold q_{thresh}. Every element with a quality $q_{el} \leq q_{thresh}$ is included in a set E (Fig. 4.1a). The extraction of the local mesh configurations contours that participate in the reconnection operation rely on a Depth First Search (DFS) edge traversal of the elements in E (Fig. 4.1b). Starting from an element in E, an element that is adjacent to an edge and belongs to E is visited. The traversal continues by visiting another element of E adjacent by edge to the already visited element. If no such elements are found near the visited element, the traversal continues from another edge-neighbor element of the starting element that belongs to E . This procedure continues until no further edge adjacent elements of E can be found. A supplemental stopping criterion of the traversal procedure is that the edge number of the contour that includes all the traversed elements is $N_C < 10$; this supplemental criterion ensures the availability of trained connectivity networks for meshing small contours. If an element of E is isolated, i.e not adjacent by edge to an another of the set, then the edge adjacent element that has the lowest quality is traversed to form an contour with $N_C = 4$ edges.



(a) (b) (c)

Figure 4.1: Example of mesh partitioning during reconnection algorithm for a square mesh. (a) Initially all elements that have a quality q_{el} lower than a quality threshold q_{thresh} (highlighted) are included in the set E ={1,2,3,4,5,6}. (b) The contours (dotted line) that are extracted after edge traversal of the elements $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ and $5 \rightarrow 6$ undergo the reconnection operation. (c) The triangulated contours after the applying the reconnection operation (Fig. 4.2).

The reconnection operation is applied to every contour that is extracted using the aforementioned procedure. Each contour is transformed using the feature transformation F (section 2.3) and meshed using the connectivity network NN_3. The connectivity information is then mapped back to the mesh using the inverse transformation F^{-1} (Fig. 4.2, Fig. 4.1c).

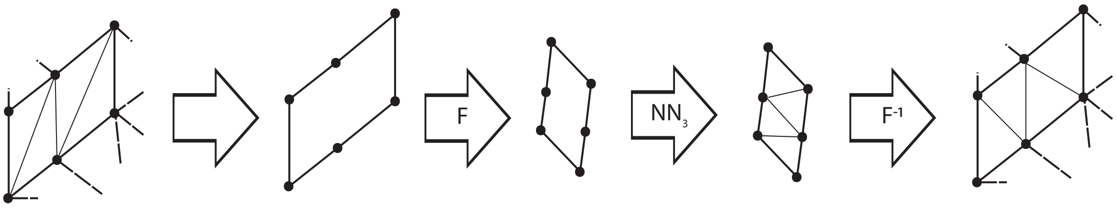
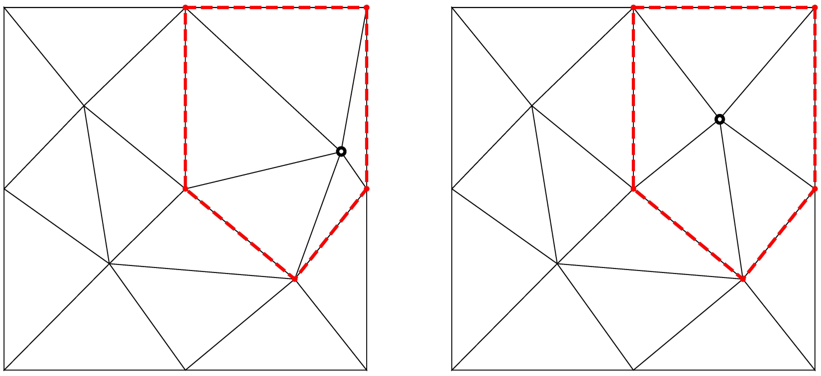


Figure 4.2: From left to right: Example of a local mesh configuration containing elements below a quality threshold. Edges of the elements are deleted creating a contour cavity. The cavity undergoes the feature transformation F and is feeded to NN_3. NN_3 outputs values of the connection table. Based on the values of the connection table the triangulation algorithm of the meshing scheme meshes the contour cavity. The connectivity information of the elements is mapped back using F^{-1} to the original mesh to complete the reconnection process.

4.1.2 Vertex Repositioning

When applying the vertex repositioning operation, the mesh is partitioned according to the vertex quality q_v. The vertex quality q_v is defined as the minimum quality q_{min} of the elements that are connected to it. If a vertex v has a quality $q_v \leq q_{thresh}$, the vertex repositioning operation is applied to the contour that includes all the elements that are connected to v (Fig. 4.3). The operation uses NN_S to predict a new optimal location of v such that the quality of the adjoined elements improves.



(a) (b)

Figure 4.3: Example of mesh partitioning during the smoothing operation. A vertex v with $q_v \leq q_{thresh}$ is spotted (a). A new optimal vertex position is predicted using NN_S for the contour including all the elements that are connected with the vertex (dotted line) (b).

The coordinates of the candidate contour are transformed by applying the transformation F. The coordinates P_C of the transformed contour cavity are given as an input to the vertex repositioning network NN_S. NN_S outputs the coordinates $\hat{p}_o = (\hat{x}_o, \hat{y}_o)$ which is an approximation of the optimal inner vertex location quality-wise. Through the inverse transformation F^{-1} the optimal inner vertex location is mapped back to the mesh (Fig. 4.4). NN_S is trained to minimize the loss function $\mathscr{L}(p_o, \hat{p}_o) = \sum(p_o - \hat{p}_o)^2/2$, where p_o is the real optimal location of the inner vertex (Fig. 4.5) (Alg.7).

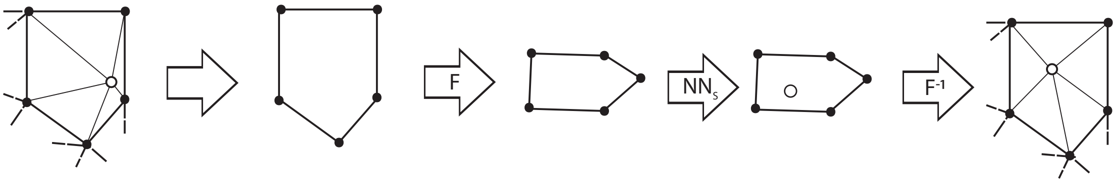


Figure 4.4: From left to right: Example of a local mesh configuration containing a vertex below a quality threshold. The elements that are connected to the vertex are deleted to create a contour cavity. The contour is transformed and the coordinates of the transformed contour are given as an input to NN_S. NN_S outputs the position of an inner vertex that is optimal quality-wise given that all contour points are connected with it. Finally, the vertex is mapped back to the mesh.

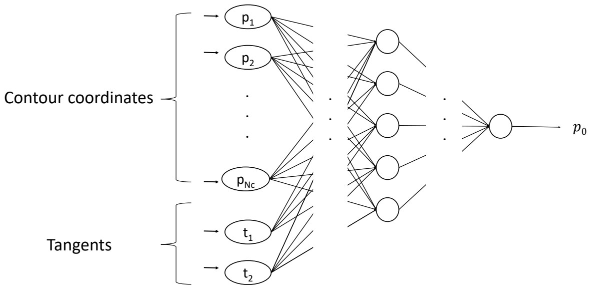


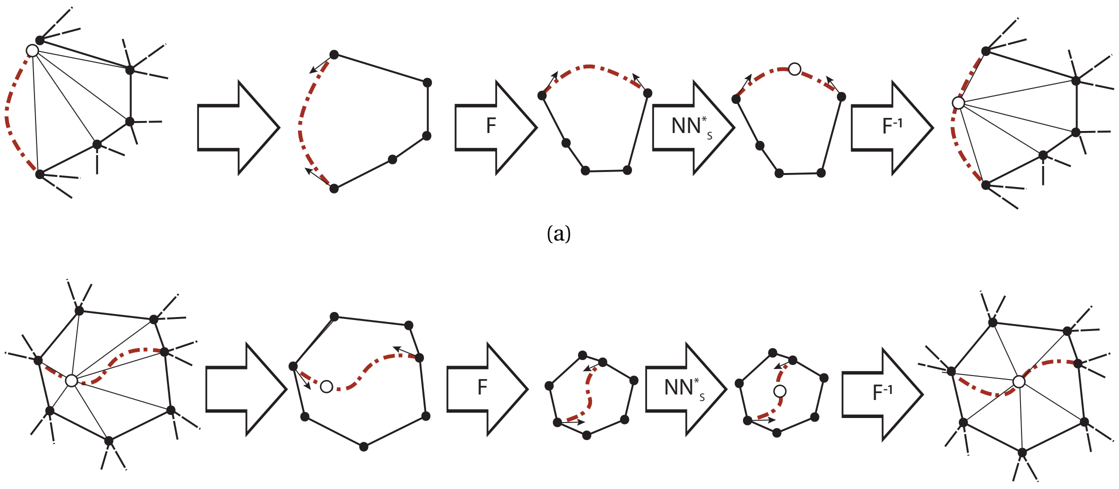
Figure 4.5: Architecture of NN_S and NN_S^*. Both are feeding forward NNs that output the coordinates $\hat{p}_o = (x_o, y_o)$ of the optimal vertex position for the vertex repositioning and surface control operations. NN_S takes as input the contour coordinates P_C whereas NN_S^* takes as input P_C and also the tangents $t = (t_1, t_2)$ of the boundary or interface the reallocated vertex belongs to.

Algorithm 7: Training algorithm of NN_S and NN_S^* for the prediction of the optimal position of an inner, boundary or interface vertex.

```
$P_C$: Contour vertices coordinates
$P_I$: Inner vertices coordinates
$p_o$: Real optimal location of inner, boundary or interface vertex
$\hat{p}_o$: Estimated optimal location of inner, boundary or interface vertex
$t = (t_1, t_2)$: Tangents of boundary or interface curve
$N_{train}$: Number of training data population
$N_{d_A}$: Dimension of flattened connection table
Initialize weights $W_K$ of $NN_S$ $(NN_S^*)$
while required number of iterations is not reached do
    foreach training example in
      $D = \{(P_C^{(n)}, p_o^{(n)})\}_i$, $(D = \{(P_C^{(n)}, p_o^{(n)}, t^{(n)})\}_i\, for\, NN_S^*)$, $i = 1, .., N_{train}$ do
        Compute $\widehat{p_o^{(n)}}$ using current parameters
        Calculate loss function
          $\mathscr{L}(p_o^{(n)}, \hat{p}_o^{(n)}) = ||p_o^{(n)} - \hat{p}_o^{(n)}||_2^2/2 = ((x_o^{(n)} - \hat{x}_o^{(n)})^2 + (y_o^{(n)} - \hat{y}_o^{(n)})^2)/2$ and $\frac{\partial \mathscr{L}}{\partial W_K}$
        Update $W_K$ using Adam learning rate optimization
    end
end
```

4.1.3 Surface control

The vertex repositioning operation has also been adapted to relocate boundary and interface vertices with constraints to respect the geometry of the boundary and interface they are part of. If the vertex belongs to the boundary, the edges of the elements containing the vertex are deleted to form an open contour. The coordinates of the open contour with the tangents $t = (t_1, t_2)$ of the boundary are transformed and given as input to NN_S^* (Fig. 4.6a). Similarly to NN_S, NN_S^* outputs the optimal position of the vertex quality-wise. The vertex position is mapped back to the mesh and projected to the boundary. Likewise, when a vertex belongs to an interface the edges of the elements containing it are deleted to form a closed contour. The aforementioned procedure is followed to the closed contour and the tangents of the interface to obtain the approximation of the optimal position of the vertex (Fig. 4.6b).



(a)

(b)

Figure 4.6: Example of local meshing configurations containing a low quality vertex that belongs to a boundary (a) and a low quality vertex that belongs to an interface (b). The edges of the element connected to the vertex are deleted to form an open contour in the case of a boundary vertex and a closed contour for the interface vertex. The contour coordinates along with tangents of the boundary or interface curve are transformed and given as an input to NN_S^*. NN_S^* outputs the optimal position of the vertex which is mapped back to the mesh. Finally, the vertex is projected to the curve.

4.1.4 Size control

The size control operations are used to adjust edge lengths close to a target edge length l_s. When applying the size control operations, the mesh is partitioned into local mesh configurations such that they contain edges that are either shorter than a shorter edge length threshold l_{thresh} or longer than a long edge length threshold L_{thresh}. To regulate the size of long edges with an edge length $l_e > L_{thresh}$, a vertex is inserted in the middle of each edge and the contour that includes their adjacent elements is extracted (Fig. 4.7). After deleting the long edges included in the contour to form a cavity, the size control operation is applied. Note that after such as partitioning of the mesh, the contour cavities could either include one inserted vertex or multiple inserted vertices if multiple long edges are located inside of it. The contour cavity with the inner vertices are transformed applying F. The transformed contour cavity and the inner vertices are then given as input to NN_3 that outputs the values of the connection table A. The triangulation algorithm uses the values to mesh the contour cavity. The connectivity information is mapped through F^{-1} to the original mesh.

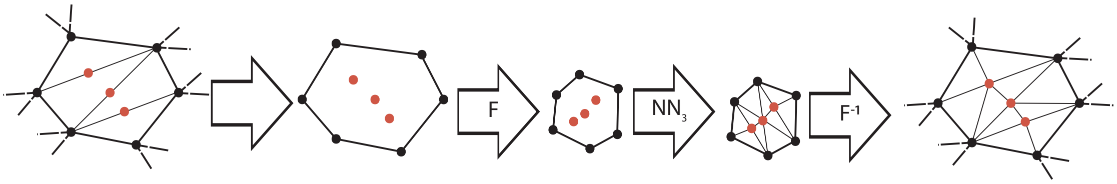


Figure 4.7: Example of a local mesh configuration containing three long edges. First, a vertex is inserted in the middle of each long edge. The long edges are deleted to form a contour cavity containing the inserted vertices. The contour with the inner vertices are meshed using NN_3. Finally, the connectivity information is mapped through F^{-1} to the mesh.

The length regulation of edges with a length $l_e \leq l_{thresh}$ starts with inserting a vertex in the middle of the edges. The contours that include the adjacent elements of the short edges as well as their neighbor elements are extracted. Subsequently, the short edges are deleted to form a contour cavity to apply the size control operation (Fig. 4.8). As in the case of partitioning the mesh to apply the size control operation for the long edges, the contour cavities may include either one vertex or several vertices if multiple short edges were included in the local mesh configuration. The contours are then meshed using the same procedure.

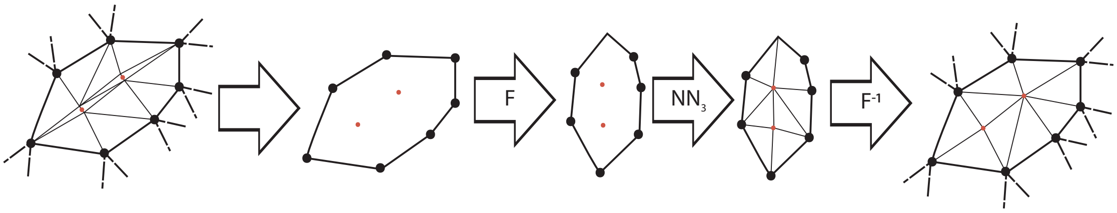


Figure 4.8: Example of a local mesh configuration containing two short edges. First, a vertex is inserted in the middle of each short edge. The elements containing the short edges and their adjacent elements are deleted to form a contour cavity containing the inserted vertices. The contour with the inner vertices are meshed using NN_3. Finally, the connectivity information is mapped through F^{-1} to the mesh.

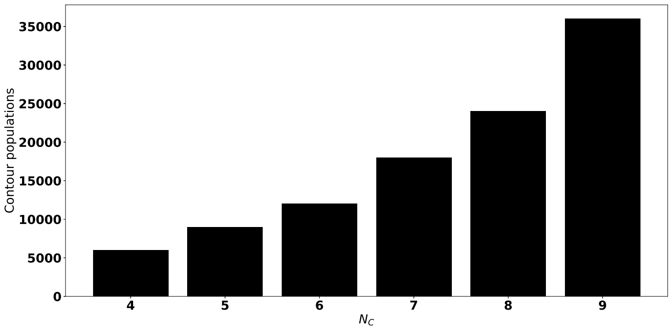
5 Results and validation of 2-D Local Mesh Improvement using Neural Networks

5.1 Experimental parameters and results

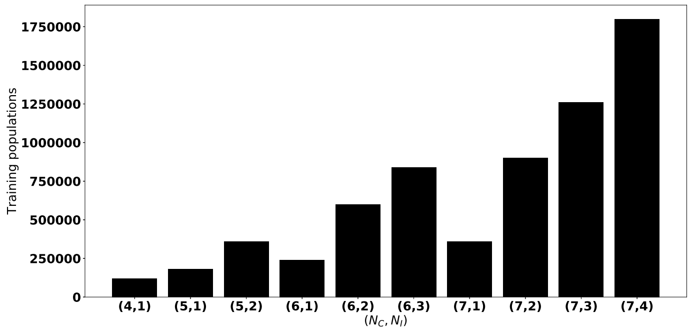
5.1.1 NN hyperparameters and training populations

For the connectivity network NN_3 used in the reconnection operation where no inner vertices are located inside the contour cavities, the contour training datasets are populations of contours for $N_C = \{4,5,6,7,8,9\}$ edges. The population of contours that are generated, are increased at a nearly exponential rate with the number of edges (Fig. 5.1a). Similarly to the meshing scheme, this increase is necessary to retain a level of desirable accuracy from the NNs; this level of accuracy in turn ensures that the application of the local mesh operations lead to good quality meshes. The same population of contours are used to train NN_S and NN_S^*.

When applying the size control operations, NN_3 is used to output the values of connection table A for contour cavities that contain inner vertices. The training of NN_3 to predict the connectivity of a contour cavity with N_I inner vertices revolves around a process of data augmentation; for every contour, the connection table A is calculated for multiple N_I-pairs of inner vertices that are sampled in the interior of the contour cavity and is included in the training dataset. For $N_I = \{1,2,3,4\}$ inner vertices 10, 20, 30, and 50 pairs of N_I inner vertices are sampled for the initial contour population of $N_C = \{4,5,6,7\}$ edges. This process of data augmentation results in the training populations shown in Fig. 5.1b.



(a)



(b)

Figure 5.1: (a) The contour populations for training NN_3, NN_S and NN_S^*. The population of 6000 contours with 4 edges is found to be an adequate training set for acquiring good quality mesh improvement results. To retain the same level of accuracy the contour populations are increased at at a nearly exponential rate with the number of N_C edges. (b) The initial contour populations of $N_C = \{4,5,6,7\}$ is augmented by sampling 10,20,30 and 50 N_I-pairs of inner vertices for $N_I = \{1,2,3,4\}$ respectively. The training pairs (N_C, N_I) are used to train NN_3 for the application of the size control operations to mesh the contour cavities with the inner vertices that are created after partitioning the mesh.

NN_S and NN_S^* are trained for 1000 epochs with a learning rate of $\eta = 10^{-4}$ and a weight decay $\lambda = 10^{-1}$ with a batch size $n_{batch} = 512$. There are 5 layers with batch normalization and the ReLU activation function. The number of hidden nodes is $(N_C + 2)^2$ for NN_S, $(N_C + 4)^2$ and $(N_C + 2)^2$ for NN_S^* when used for interface and boundary vertex repositioning respectively, where N_C is the number of contour edges. The hyperparameters and architecture of NN_3 are as presented in section 3.1.3.

5.1.2 Experiments

The proposed operations are tested for static and dynamic mesh improvement on a variety of meshes using the quality metric of section 2.6.1 (eq 2.1). During the static case studies, edges of the mesh are randomly flipped and the vertices are perturbed such that low quality elements are produced. Unless stated otherwise, the reconnection operation is followed by vertex repositioning to improve the quality of the mesh using a quality threshold $q_{thresh} = 0.8$. The operations are evaluated in terms of quality and angle distribution of the elements.

In the dynamic test cases, vertices of the mesh are advected according to an analytical velocity field for a total of T timesteps. At each timestep, the mesh is improved using a local improvement scheme that includes the operations. The operations are reviewed in terms of minimum quality, minimum and maximum angles before and after applying the scheme. The scheme (Alg. 8) starts by applying the vertex repositioning and surface control operation to every vertex with a quality lower than q_{tresh} (Alg. 8, Line 2). The reconnection operation is successively applied to every element having a quality below q_{thresh} (Alg. 8, Line 3). As a next step, the length of the edges are regulated according to a given target edge length l_s. The size control operation is applied for long edges with a length larger than $L_{thresh} = 4/3 l_s$ (Alg. 8, Line 4) and for short edges with a length shorter than $l_{thresh} = 4/5 l_s$ (Alg. 8, Line 4). If the minimum quality of the mesh q_{worst} is lower than q_{thresh}, the scheme applies the vertex repositioning and surface control operations followed by the reconnection operation until either $q_{worst} \geq q_{thresh}$ or a number of maximum iterations *maxNumOfPasses* is exceeded (Alg. 8, Lines 6-14). A number of *maxNumOfPasses* = 5 iterations and a quality threshold of $q_{thresh} = 0.8$ are chosen.

Algorithm 8: The Local Improvement Scheme for the dynamic meshes.

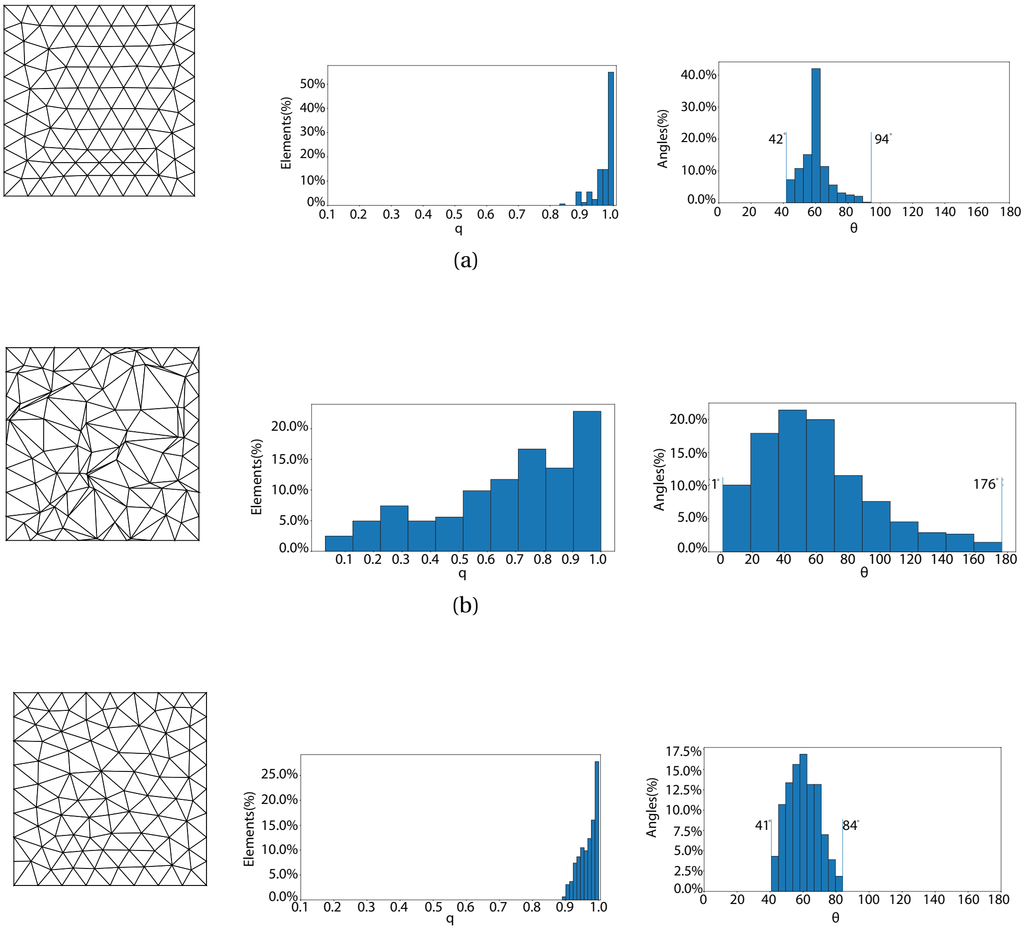
```
1  ImprovementScheme (maxNumOfPasses, $q_{thresh}$, $l_s$)
2      Apply vertex repositioning and surface control operations for every vertex $v$ with
         vertex quality $q_v \leq q_{thresh}$
3      Apply reconnection operation for every element $e$ with quality $q_e \leq q_{thresh}$
4      Apply size control operation for every edge that is longer than $L_{threshold} = 4/3 l_s$
5      Apply size control operation for every edge that is shorter than $l_{threshold} = 4/5 l_s$
6      while numOfPasses <maxNumOfPasses do
7          if $q_{worst} \geq q_{thresh}$ then
8              return
9          end
10         else
11             Apply vertex repositioning and surface control operations
12             Apply reconnection operation
13         end
14     end
```

All initial meshes were generated using the *Gmsh*© software that applies a CDT algorithm (Lambrechts et al. (2008)) followed by refinement. The sizing of the elements is calculated via interpolation of target edge values assigned to the vertices of the initial geometry (see **Appendix A.2.2**).

5.1.2.1 Static Mesh Improvement

Square

Mesh improvement using the operations is tested for a square mesh in a domain $\Omega = [0,1] \times [0,1]$ with mean quality $q_{mean} = 0.97$, minimum quality $q_{min} = 0.83$, angles that lie between $43° \leq \theta \leq 94°$ (Fig 5.2a) and a uniform target edge length $l_s = 0.12$. After edge flipping and vertex pertubation, the mean quality decreases to $q_{mean} = 0.63$, the minimum quality decreases to $q_{worst} = 0.05$, and the element angle distribution lies between $1° \leq \theta \leq 176°$ (Fig 5.2b). By applying the reconnection and vertex repositioning operations, the mean quality increases to $q_{mean} = 0.96$, the minimum quality increases to $q_{worst} = 0.89$, and the element angle distribution lies between $41° \leq \theta \leq 84°$ (Fig 5.2c).



(a)

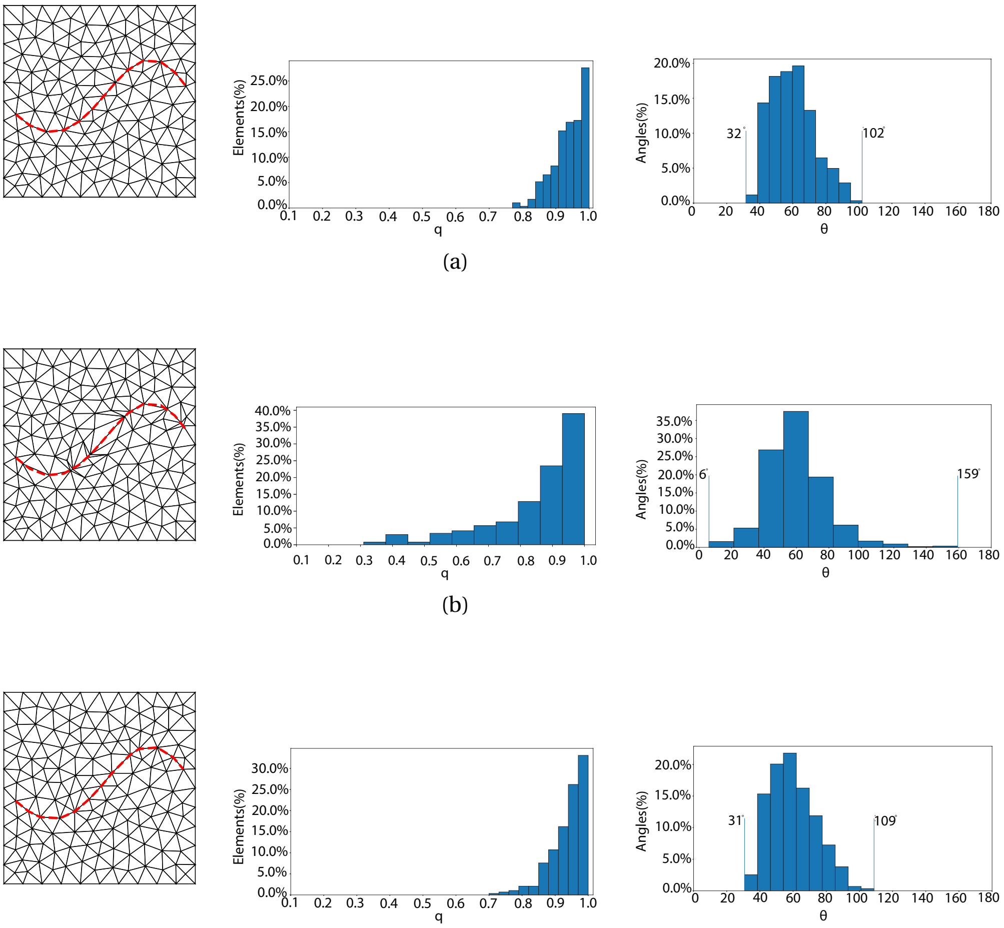
(b)

(c)

Figure 5.2: Example of mesh improvement for a meshed square in a domain $\Omega = [0,1] \times [0,1]$. (a) Initially the mesh has a mean quality $q_{mean} = 0.97$, a minimum quality $q_{worst} = 0.83$, and a element angle distribution ranging between $43° \leq \theta \leq 94°$. (b) After random edge flipping and vertex perturbation the mean quality of the mesh decreases to $q_{mean} = 0.63$, the minimum quality decreases to $q_{worst} = 0.05$, and the element angles lie between $1° \leq \theta \leq 176°$. (c) After applying the reconnection and vertex repositioning operations, the mean quality of the mesh increases to $q_{mean} = 0.96$, the minimum quality $q_{worst} = 0.89$, and element angles lie between $41° \leq \theta \leq 84°$.

Perturbed interface

The operation of interface surface control is tested in a meshed square on a domain $\Omega = [0,1] \times [0,1]$ containing an interior interface curve. Initially, the mesh has $q_{mean} = 0.93$, $q_{worst} = 0.77$, element angles ranging between $32° \leq \theta \leq 102°$ (Fig. 5.3a) and a uniform target edge length $l_s = 0.1$. The vertices of the interface are perturbed to produce low quality elements near the interface lowering the mean quality to $q_{mean} = 0.68$, the minimum quality to $q_{wost} = 0.32$, and causing element angles to lie between 6° $\leq \theta \leq$ 159° (Fig. 5.3b). The interfaces vertices are then repositioned using the surface control operation. After applying the operation, the mean quality of the mesh increases to $q_{mean} = 0.92$, the minimum quality increases to $q_{worst} = 0.7$, and the element angles lie between 31° $\leq \theta \leq$ 109° (Fig. 5.3c).



(a)

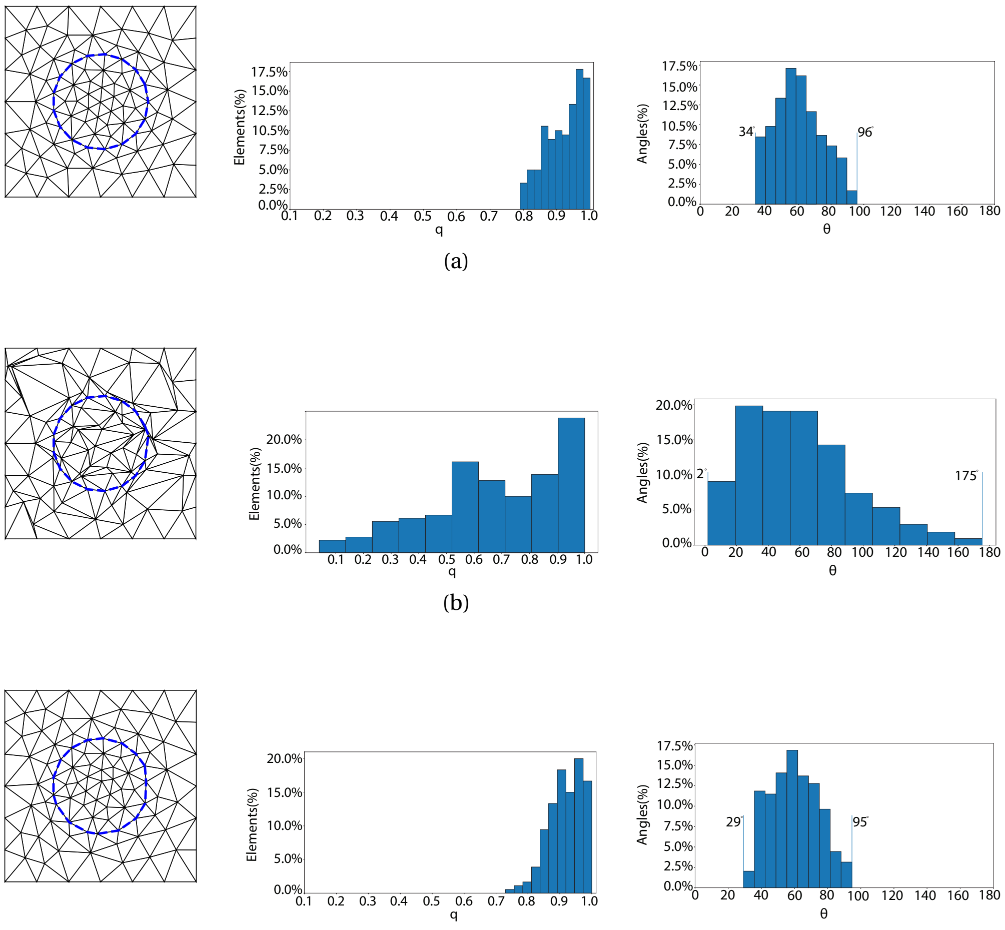
(b)

(c)

Figure 5.3: Example of applying the surface control operation to the interface vertices of a curve (highlighted) of a square mesh in a domain $\Omega = [0,1] \times [0,1]$. (a) Initially the mesh has a mean quality $q_{mean} = 0.93$, a minimum quality $q_{worst} = 0.77$, and a element angle distribution lying between 32° $\leq \theta \leq$ 102°. (b) After perturbing the interface vertices, the mean quality of the mesh decreases to $q_{mean} = 0.68$, the minimum quality decreases to $q_{worst} = 0.32$, and the element angles lie between 6° $\leq \theta \leq$ 159°. (c) After applying the surface control operation to the interface vertices, the mean quality of the mesh increases to $q_{mean} = 0.92$, the minimum quality increases to $q_{worst} = 0.7$, and the element angles range between 31° $\leq \theta \leq$ 109°.

Circular interface

A mesh of a square domain $\Omega = [0,1] \times [0,1]$ contains a circular interface with $q_{mean} = 0.93$, $q_{worst} = 0.78$ and element angles that lie between $34° \leq \theta \leq 96°$ (Fig 5.4a). The vertices on the boundary of the mesh are assigned a target edge length value $l_s = 0.16$, while the vertices that belong to the circular interface are assigned a target edge length value $l_s^I = 0.08$. The size of the mesh elements is then computed by interpolating these values inside the domain during mesh generation. The mesh has its edges randomly flipped and vertices perturbed; as a result, the mean quality decreases to $q_{mean} = 0.67$, the minimum quality decreases to $q_{worst} = 0.04$ and the element angles range between $2° \leq \theta \leq 175°$ (Fig 5.4b). After applying the operations of reconnection, vertex repositioning for the interior, and surface control for the interface and boundary vertices, the mean quality increases to $q_{mean} = 0.92$, the minimum quality increases to $q_{worst} = 0.74$ and element angles range between $29° \leq \theta \leq 95°$ (Fig 5.4c).



(a)

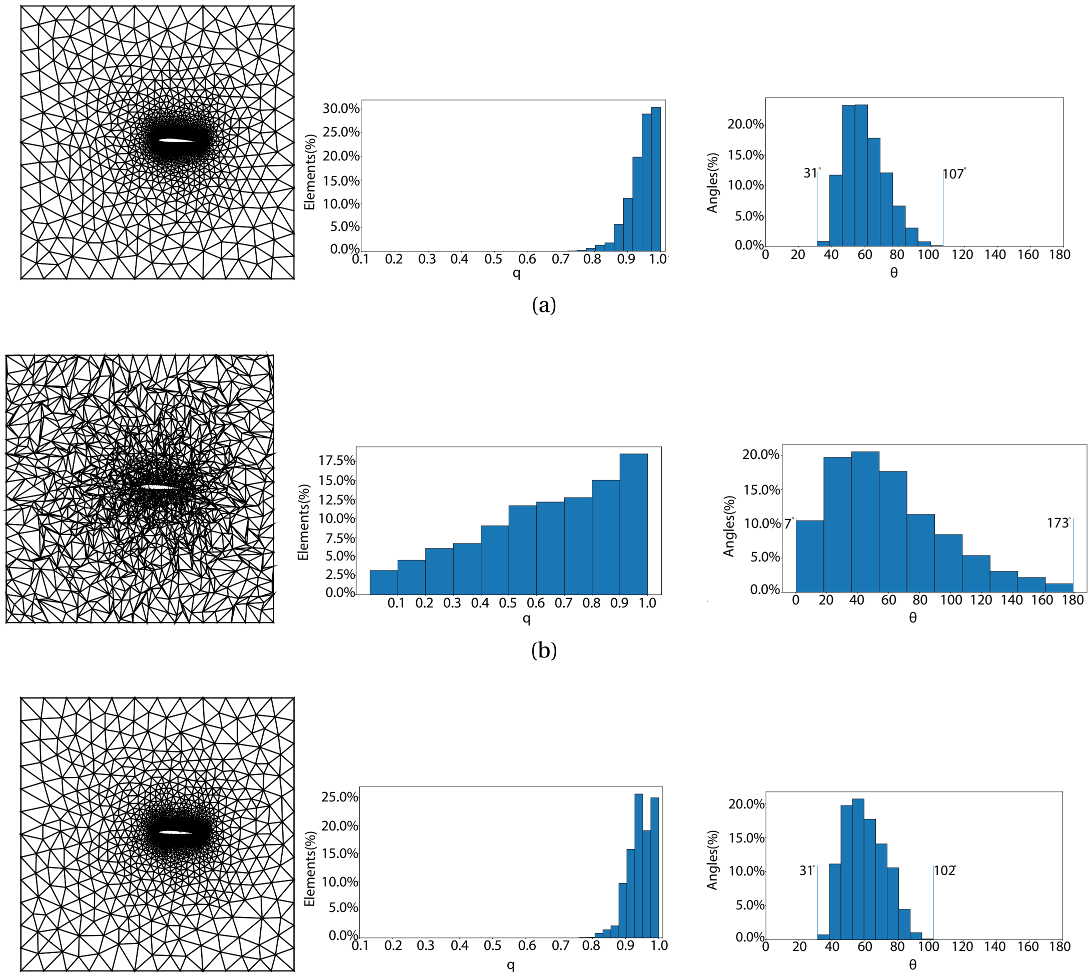
(b)

(c)

Figure 5.4: Example of improving the quality of a square mesh in a domain $\Omega = [0,1] \times [0,1]$ containing a circular interface in the center (highlighted). (a) Initially the mesh has a mean quality $q_{mean} = 0.93$, a minimum quality $q_{worst} = 0.78$, and a element angle distribution ranging between $34° \leq \theta \leq 96°$. (b) After randomly flipping the edges and perturbing the vertices, the mean and minimum quality of the mesh decrease to $q_{mean} = 0.67$ and $q_{worst} = 0.04$, respectively, with element angles lying between $2° \leq \theta \leq 175°$. (c) After applying the reconnection operation followed by the vertex repositioning operation to the vertices to the perturbed mesh, the mean quality of the mesh increases to $q_{mean} = 0.92$, the minimum quality increases to $q_{worst} = 0.74$, and element angles lie between $29° \leq \theta \leq 95°$.

Airfoil

A mesh domain $\Omega = [-3,3] \times [-3,3]$ represents the exterior domain of an airfoil (hole) with mean quality $q_{mean} = 0.94$, minimum quality $q_{worst} = 0.74$, and elements that lie between $32° \leq \theta \leq 107°$ (Fig. 5.5a). The element size of the mesh is calculated by means of interpolating the values of the boundary vertices with a prescribed target edge length value $l_s = 0.2$ and the values of the airfoil interface vertices with a prescribed target edge length value $l_s^I = 0.002$. The vertices of the mesh are perturbed and the edges are randomly flipped. As a result, the mean quality decreases to $q_{mean} = 0.64$, the minimum quality decreases to $q_{worst} = 0.03$, and the element angles lie between $7^\circ \leq \theta \leq 173^\circ$ (Fig. 5.5b). The operations of reconnection followed by vertex repositioning of interior vertices and surface control of interface vertices are applied to improve the quality of the perturbed mesh. The mean quality increases to $q_{mean} = 0.94$, the minimum quality increases to $q_{worst} = 0.76$ and the element angles lie between $31^\circ \leq \theta \leq 102^\circ$ (Fig. 5.5c).



(a)

(b)

(c)

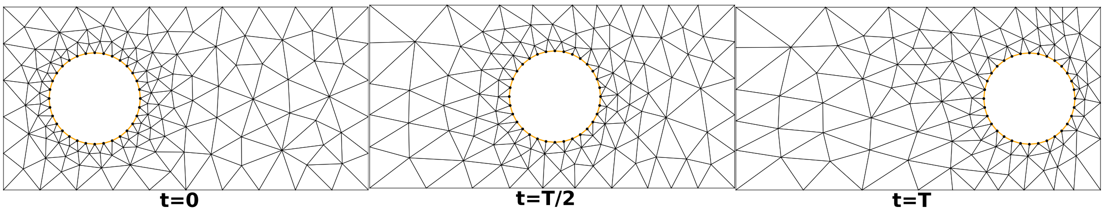
Figure 5.5: Example of applying the mesh improvement operations of a square mesh in a domain $\Omega = [0,1] \times [0,1]$ containing an airfoil shaped hole. (a) Initially the mesh has a mean quality $q_{mean} = 0.91$, a minimum quality $q_{worst} = 0.74$, and a element angle distribution lying between $32° \leq \theta \leq 107°$. (b) After perturbing the vertices and flipping the edges, the mean quality of the mesh decreases to $q_{mean} = 0.64$, the minimum quality decreases to $q_{worst} = 0.03$, and element angles lie between $7° \leq \theta \leq 173°$. (c) After applying the reconnection operation followed by vertex repositioning and surface control, the mean quality of the mesh increases to $q_{mean} = 0.94$, the minimum quality increases to $q_{worst} = 0.76$, and the element angles lie between $31° \leq \theta \leq 102°$.

5.1.2.2 Dynamic Mesh Improvement

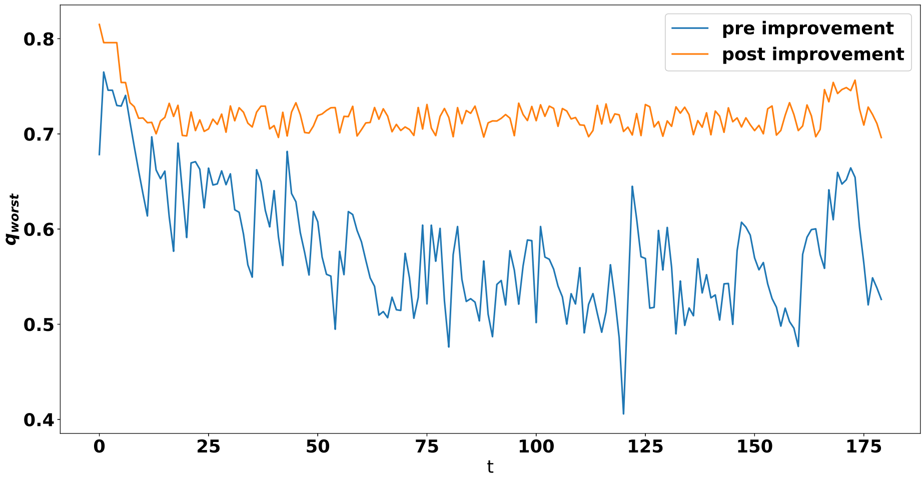
Horizontal translation of circular interface

The vertices of a circular interface with radius $r = 0.25$ inside a rectangle mesh with domain $\Omega = [0,2] \times [0,1]$ are translated at an horizontal velocity of $v_x = 5 \cdot 10^{-3} \hat{x}$, where $\hat{x}$ is the unit vector along the x axis, for a total of $T = 180$ timesteps. The vertices on the boundary of the mesh are assigned a target edge length value $l_s = 0.2$, while the vertices that belong to the circular interface are assigned a target edge length value $l_s^I = 0.05$. The size of the mesh elements is then computed by interpolating these values inside the domain during mesh generation. The center of the circular interface is initially located at $p_o = (0.5, 0.5)$ and after $t = 180$ time steps its final position is $p_f = (1.4, 0.5)$ (Fig. 5.6a-c). At each step of the simulation, the local mesh improvement scheme (Alg. 8) applies the vertex repositioning operation to the interior vertices of the mesh and the surface control operation to the interface (circle) and boundary (rectangle) vertices. The reconnection is applied next. The scheme subsequently applies the size control operations to retain the edge length difference between the elements close to the interface and the elements elsewhere. Finally, the scheme applies up to 5 iterations of vertex repositioning and surface control followed by reconnection.

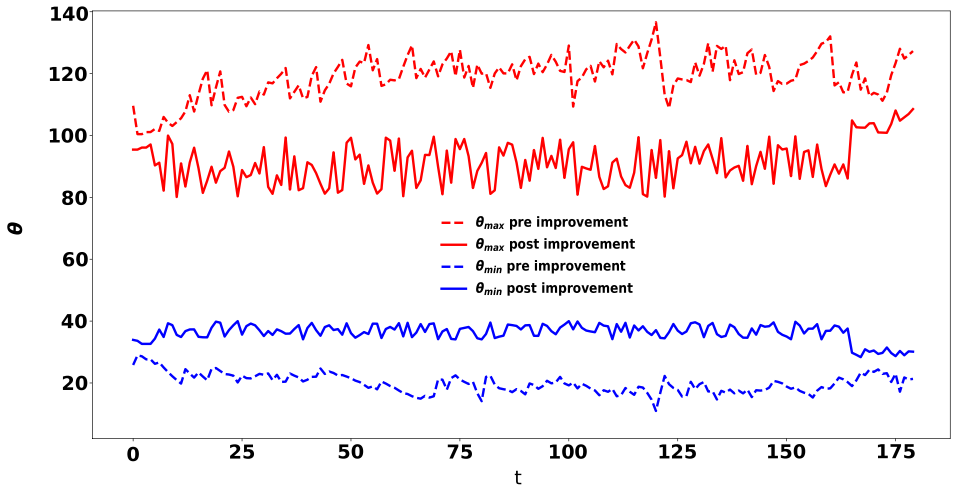
The minimum element quality q_{worst} of the mesh before applying the local improvement scheme lies between $0.41 \le q_{worst} \le 0.76$ (Fig.5.6d), the minimum element angles lie between $10 \le \theta_{min} \le 28$, and the maximum angles lie between $94 \le \theta_{max} \le 133$ (Fig.5.6e). At each simulation step, the scheme improves q_{worst} which lies between $0.69 \le q_{worst} \le 0.81$, increases the minimum element angles that lie between $28 \le \theta_{min} \le 40$, and decreases the maximum angles that lie between $80 \le \theta_{max} \le 108$.



(a) (b) (c)



(d)



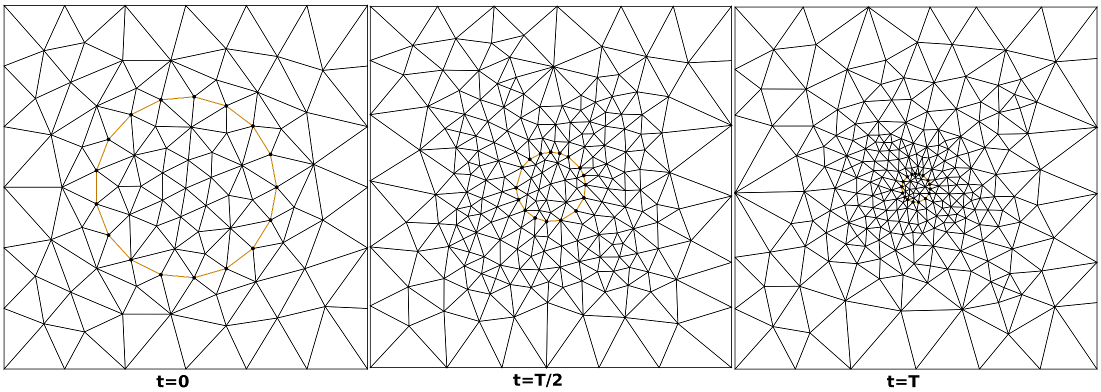
(e)

Figure 5.6: (a)-(c): The circular interface's horizontal translation for different timesteps t. (d): The minimum quality q_{worst} as a function of the simulation timestep t. q_{worst} has a range between $0.50 \leq q_{worst} \leq 0.79$ before the application of the mesh improvement scheme and $0.69 \leq q_{worst} \leq 0.81$ after applying the scheme. (e) The minimum angles lie between $10 \leq \theta_{min} \leq 28$ before the application of the scheme and $28 \leq \theta_{min} \leq 40$ after the application. The maximum angles lie between $94 \leq \theta_{max} \leq 133$ before the application of the scheme and $80 \leq \theta_{max} \leq 108$ after the application.

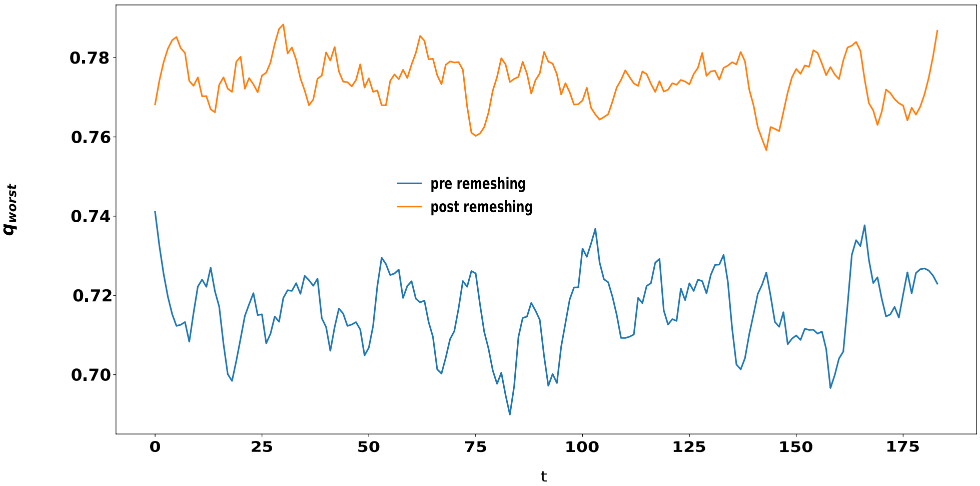
Collapsing Circle

The vertices of a circular interface with radius $r_0 = 0.25$ and center $p = (0,0)$ inside a square mesh domain $\Omega = [0,1] \times [0,1]$ (Fig. 5.7a) are translated towards the center of the circle at a constant velocity of $v_r = -1 \cdot 10^{-2}\hat{r}$, where $\hat{r}$ is the unit vector along the circle radius direction. After a total of $T = 184$ timesteps, the radius of the circle is reduced to $r_f = 0.07$. The vertices on the boundary of the mesh are assigned a target edge length value $l_s = 0.16$, while the vertices that belong to the circular interface are assigned an initial target edge length value $l_s^I = 0.08$ (Fig. 5.7c). The size of the mesh elements is then computed by interpolating these values inside the domain during mesh generation. At each step of the simulation, the edge length of the elements located around and inside the circle is adapted to the edge length of the interface; the edge length of the elements whose vertices are within a distance of $d = r_0 = 0.25$ from the circular interface vertices is adapted according to the target edge threshold of the interface edge length, while the elements elsewhere retain their initial target edge length.

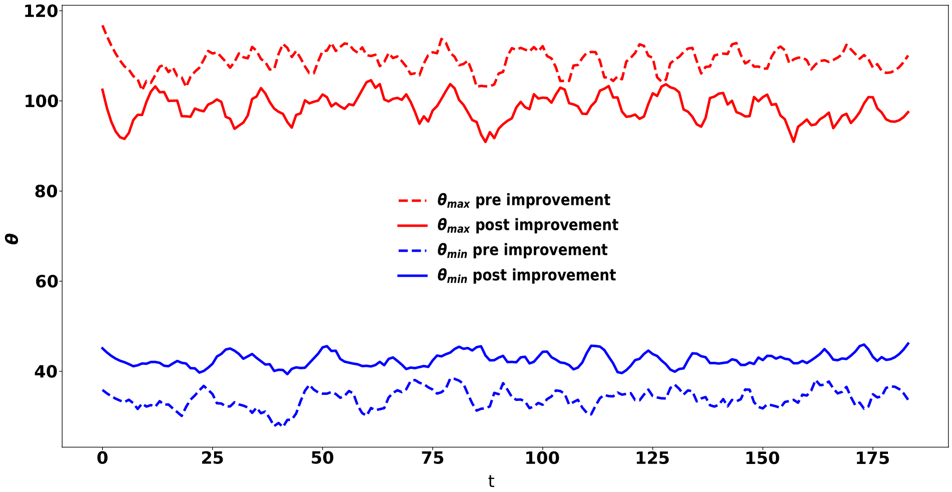
The minimum quality of the mesh q_{worst} lies between $0.68 \leq q_{worst} \leq 0.74$ before applying the mesh improvement scheme (Alg. 8) and $0.75 \leq q_{worst} \leq 0.79$ after its application (Fig. 5.7d). The minimum angles lie between $27 \leq \theta_{min} \leq 38$ before the application of scheme and $39 \leq \theta_{min} \leq 46$ after the application. The maximum angles lie between $108 \leq \theta_{max} \leq 116$ before the application of scheme and $90 \leq \theta_{max} \leq 104$ after the application (Fig. 5.7e).


t=0
t=T/2
t=T

(a) (b) (c)


0.78
0.76
0.74
0.72
0.70
q_{worst}
pre remeshing
post remeshing
0
25
50
75
100
125
150
175
t

(d)


120
100
80
60
40
θ
θ_{max} pre improvement
θ_{max} post improvement
θ_{min} pre improvement
θ_{min} post improvement
0
25
50
75
100
125
150
175
t

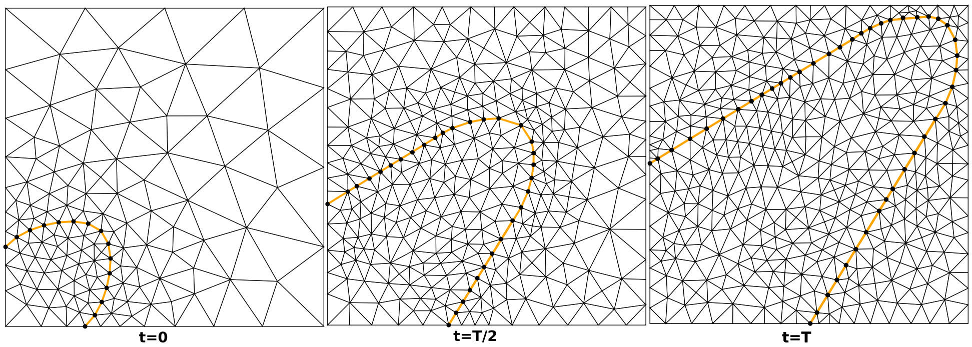
(e)

Figure 5.7: (a)-(c): The collapsing circle for different timesteps. (d):The minimum quality q_{worst} as a function of the simulation timestep t. q_{worst} lies between $0.68 \leq q_{worst} \leq 0.74$ before the application of the scheme and $0.75 \leq q_{worst} \leq 0.79$ after the application. (e) The minimum angles lie between $27 \leq \theta_{min} \leq 38$ before the application of the scheme and $39 \leq \theta_{min} \leq 46$ after the application. The maximum angles lie between $108 \leq \theta_{max} \leq 116$ before the application of the scheme and $90 \leq \theta_{max} \leq 104$ after the application.

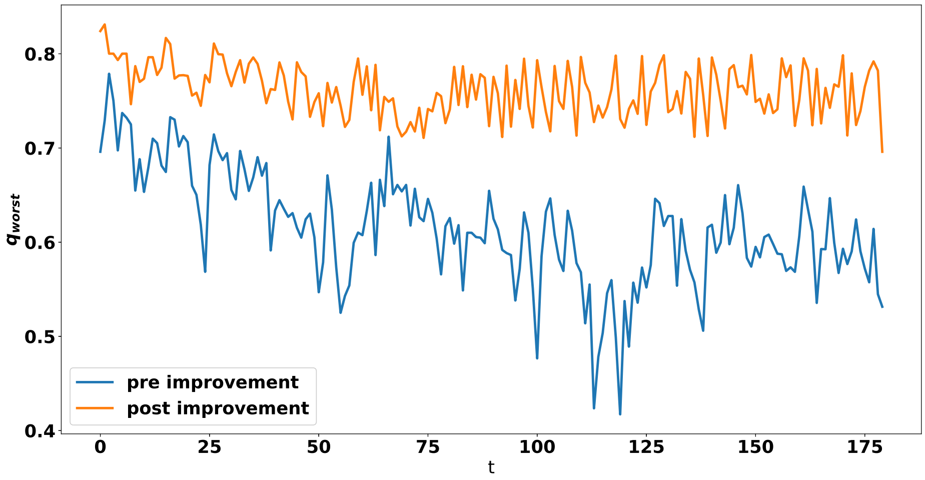
Diagonal translation of parabolic interface

The vertices of a parabolic interface inside a square mesh domain $\Omega = [-1,1] \times [-1,1]$ are translated diagonally at a constant speed $v = 10^{-2}\hat{x} + 10^{-2}\hat{y}$, where $\hat{x}$ and $\hat{y}$ are the unit vectors along the x and y axis directions respectively. The simulation is iterated for $T = 180$ timesteps (Fig. 5.8a-c). The element size of the mesh is calculated by means of interpolating the values of the boundary vertices with a prescribed target edge length value $l_s = 0.5$ and the values of the parabolic interface vertices with a prescribed target edge length value $l_s^I = 0.1$. At each step of the simulation the size control operations of the scheme (Alg. 8) are applied to retain the initial edge length of the interfaces; vertices are inserted at the midpoint of an interface edge when its length is bigger $L_{thresh} = 4/3 l_s^I$. Similarly, the size control operation for long edge is applied to all the edges of elements whose vertices are within a distance of $d = 2 \cdot 10^{-2}$ from vertices of the parabolic interface.

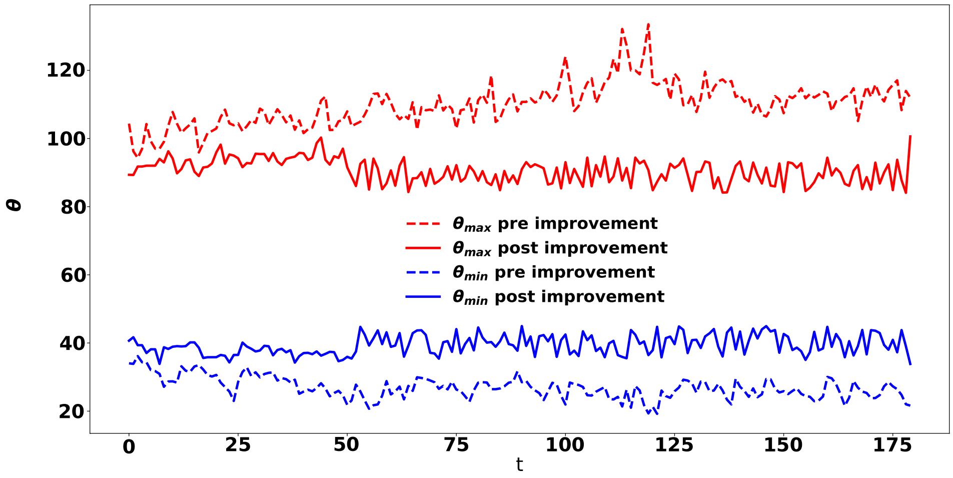
The minimum quality of the mesh q_{worst} lies between $0.41 \leq q_{worst} \leq 0.77$ before applying the meshing improvement scheme and $0.69 \leq q_{worst} \leq 0.83$ after its application (Fig. 5.8d). The minimum angles lie between $19 \leq \theta_{min} \leq 36$ before the application of scheme and $33 \leq \theta_{min} \leq 45$ after the application. The maximum angles lie between $94 \leq \theta_{max} \leq 133$ before the application of scheme and $84 \leq \theta_{max} \leq 100$ after the application (Fig. 5.8e).



(a) (b) (c)



(d)



(e)

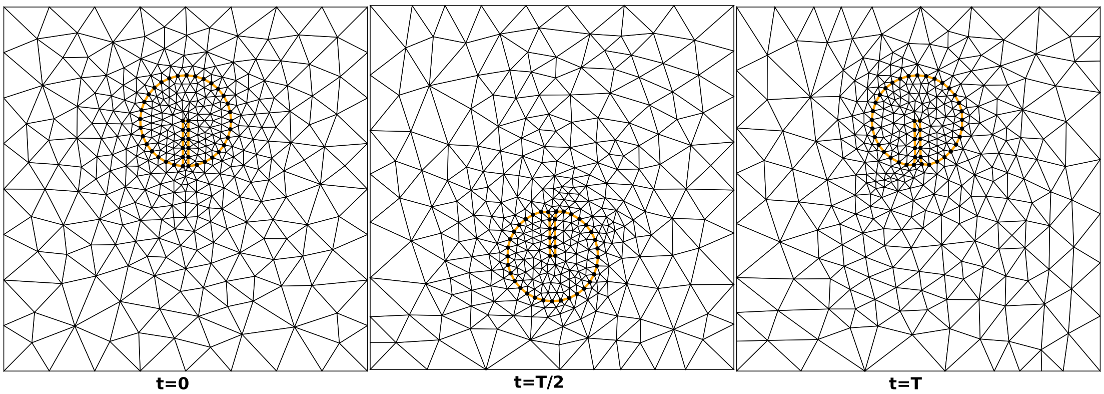
Figure 5.8: (a)-(c): Timesteps of the diagonal translation of the elliptical interface. (d): The minimum quality as a function of the simulation timestep. The minimum quality of the mesh q_{worst} lies between $0.41 \leq q_{worst} \leq 0.77$ before the application of the mesh improvement scheme and $0.69 \leq q_{worst} \leq 0.83$ after its application. (e) The minimum angles lie between $19 \leq \theta_{min} \leq 36$ before the application of the scheme and $33 \leq \theta_{min} \leq 45$ after the application. The maximum angles lie between $94 \leq \theta_{max} \leq 133$ before the application of the scheme and $84 \leq \theta_{max} \leq 100$ after the application.

Zalesak disc

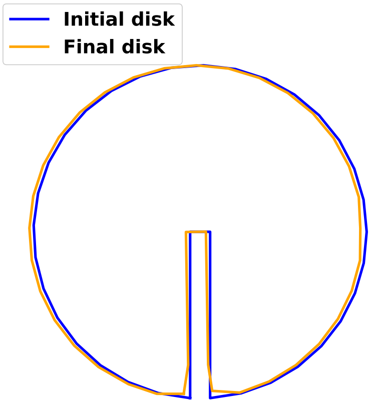
The interface vertices of a slotted disk located inside a square domain $\Omega = [0,4] \times [0,4]$ are moved according to a rotational velocity field $v = -\omega(y - y_o)\hat{x} + \omega(x - x_o)\hat{y}$, where $(x_o, y_o) = (2,2)$

is the center of rotation , $\omega = 3 \cdot 10^{-2}$ is the angular velocity, and $\hat{x}$, $\hat{y}$ are the unit vectors along the x and y axis directions respectively. The slotted disk undergoes a full rotation after $T = 210$ time steps. The vertices on the boundary of the mesh are assigned a target edge length value $l_s = 0.5$, while the vertices that belong to the slotted disc interface are assigned a target edge length value $l_s^I = 0.1$. The size of the mesh elements is then computed by interpolating these values inside the domain during mesh generation. The full rotation of the slotted disk causes the deformation of the initial interface (Fig. 5.9d) as the interface vertices are free to move to test the efficiency of the surface control operation. At each time step, the size control operations of the scheme (Alg.8) regulate the edge lengths of the elements located surrounding the interface as the disk rotates.

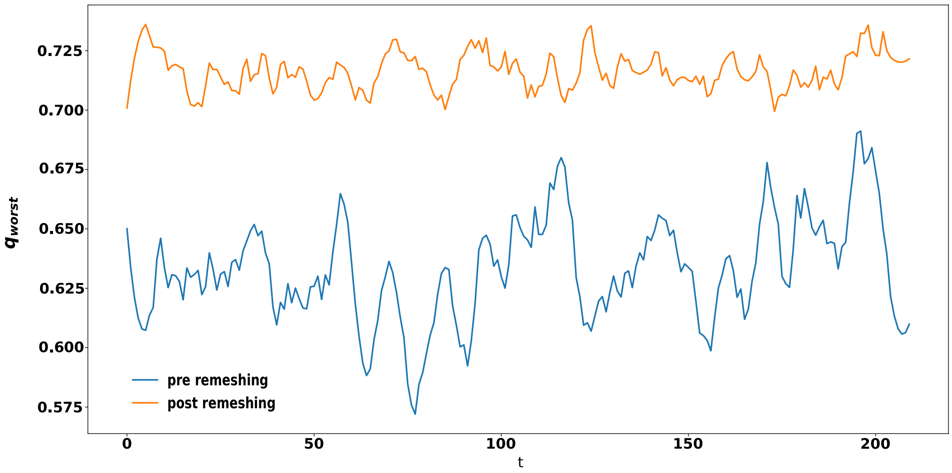
The minimum quality of the mesh q_{worst} lies between $0.58 \leq q_{worst} \leq 0.69$ before applying the mesh improvement scheme and $0.68 \leq q_{worst} \leq 0.74$ after its application (Fig. 5.9e). The minimum angles lie between $26 \leq \theta_{min} \leq 32$ before the application of scheme and $35 \leq \theta_{min} \leq 44$ after the application. The maximum angles lie between $99 \leq \theta_{max} \leq 117$ before the application of scheme and $90 \leq \theta_{max} \leq 107$ after the application (Fig. 5.8f).



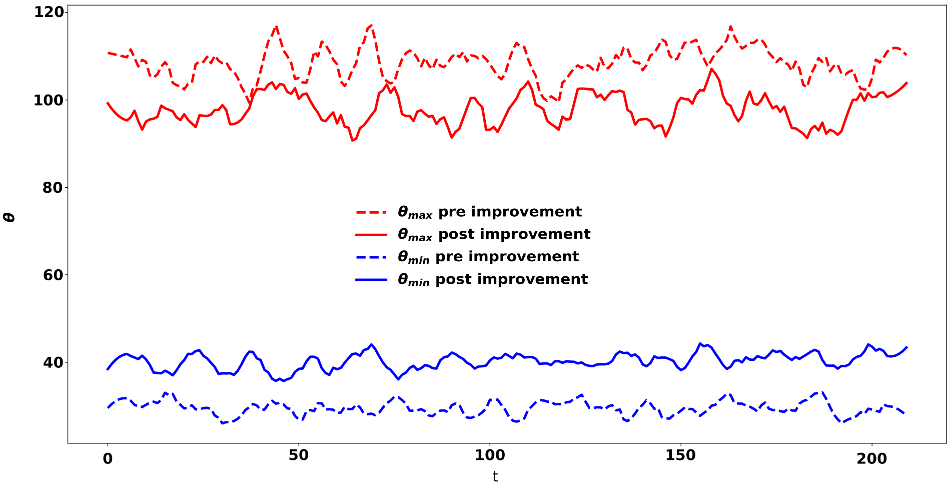
(a) (b) (c)


Initial disk
Final disk

(d)


q_{worst}
0.725
0.700
0.675
0.650
0.625
0.600
0.575
pre remeshing
post remeshing
0
50
100
150
200
t

(e)


120
100
80
60
40
θ
θ_{max} pre improvement
θ_{max} post improvement
θ_{min} pre improvement
θ_{min} post improvement
0
50
100
150
200
t

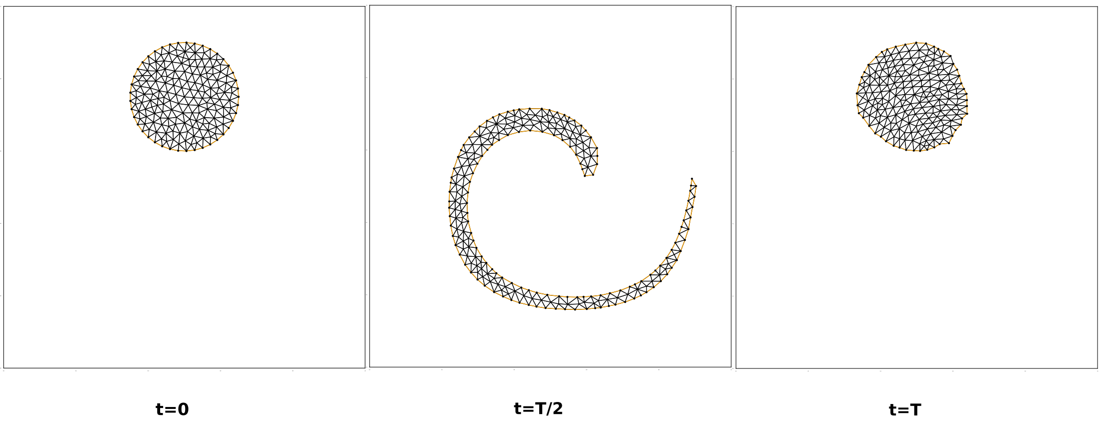
(f)

Figure 5.9: (a)-(c): Timesteps of the zalesak disc rotation. (d): The initial interface and the deformed interface after the completion of the rotation. (e): The minimum quality as a function of the simulation timestep. q_{worst} lies between $0.58 \leq q_{worst} \leq 0.69$ before applying the mesh improvement scheme and $0.68 \leq q_{worst} \leq 0.74$ after its application. (f) The minimum angles lie between $26 \leq \theta_{min} \leq 32$ before the application of the scheme and $35 \leq \theta_{min} \leq 44$ after the application. The maximum angles lie between $99 \leq \theta_{max} \leq 117$ before the application of the scheme and $90 \leq \theta_{max} \leq 107$ after the application.

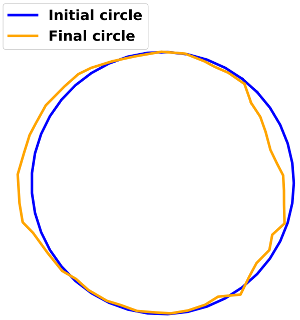
Vortex flow deformation

A mesh of a circle of radius $r = 0.15$ is centered at $(0,75)$ on a domain $\Omega = [0,1] \times [0,1]$ (Fig. 5.10a) with a uniform target edge length $l_s = 0.02$. The vertices of the circle's boundary are stretched by the velocity field $v(t) = -\cos(\pi t/\Delta T)\sin^2(\pi x)\cos(2\pi y)\hat{x} + \cos(\pi t/\Delta T)\sin^2(\pi y)\cos(2\pi x)\hat{y}$, where $\Delta T = 4$ is the assigned period and $\hat{x}$, $\hat{y}$ are the unit vectors along the x and y axis directions respectively. The circular boundary reaches maximum deformation on $t = \Delta T/2$ (Fig. 5.10b) and returns to its original position on $t = \Delta T$ (Fig. 5.10c). At $t = T$, due to the application of the remeshing operations the circular boundary is deformed when compared to its original state (Fig. 5.10d).

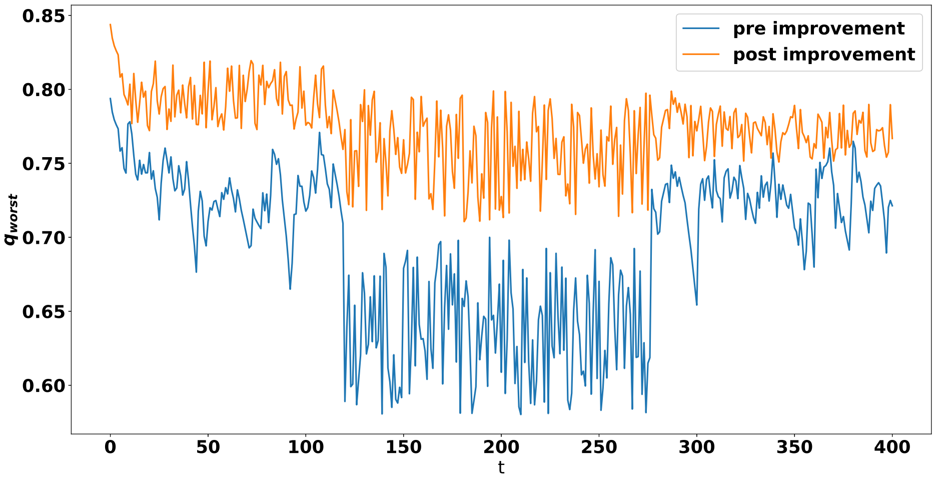
The minimum quality of the mesh q_{worst} lies between $0.58 \leq q_{worst} \leq 0.79$ before the applications of the mesh improvement scheme and $0.71 \leq q_{worst} \leq 0.84$ after the application (Fig. 5.10e). The minimum angles lie between $21 \leq \theta_{min} \leq 32$ before the application of scheme and $27 \leq \theta_{min} \leq 37$ after the application. The maximum angles lie between $98 \leq \theta_{max} \leq 120$ before the application of scheme and $90 \leq \theta_{max} \leq 108$ after the application (Fig. 5.10f).



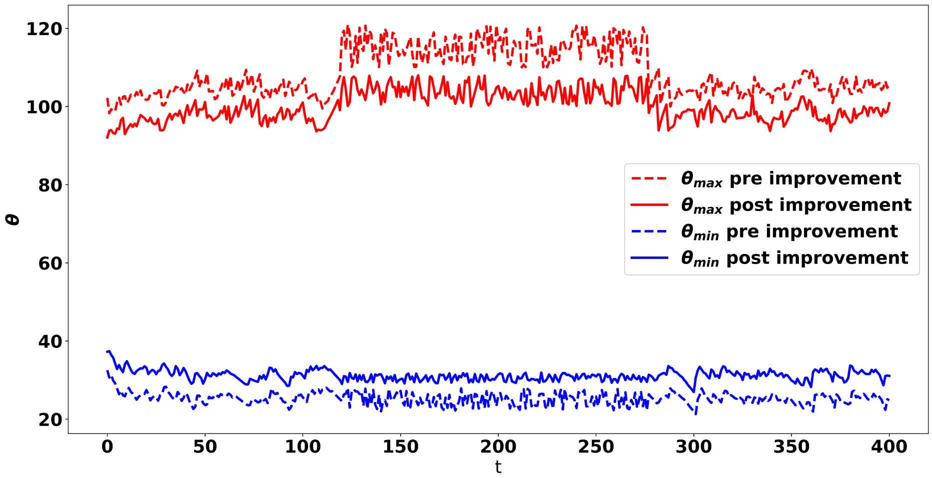
(a) (b) (c)


Initial circle
Final circle

(d)


0.85
0.80
0.75
0.70
0.65
0.60
q_{worst}
0
50
100
150
200
250
300
350
400
t
pre improvement
post improvement

(e)


120
100
80
60
40
20
θ
0
50
100
150
200
250
300
350
400
t
θ_{max} pre improvement
θ_{max} post improvement
θ_{min} pre improvement
θ_{min} post improvement

(f)

Figure 5.10: (a)-(c): Timesteps of the vortex flow deformation of the circular interface. (d): The initial interface and the deformed interface after the completion of the simulation. (e): The minimum quality as a function of the simulation timestep. q_{worst} has a range between $0.58 \leq q_{worst} \leq 0.79$ before the application of the mesh improvement scheme and $0.71 \leq q_{worst} \leq 0.84$ after the application. (f) The minimum angles lie between $21 \leq \theta_{min} \leq 32$ before the application of the scheme and $27 \leq \theta_{min} \leq 37$ after the application. The maximum angles lie between $98 \leq \theta_{max} \leq 120$ before the application of the scheme and $90 \leq \theta_{max} \leq 108$ after the application.

5.2 Conclusions

A novel method for the development of local mesh improvement operations using NNs is presented. The operations improve the quality of elements that are below a quality threshold and regulate their edge length according to a short and long edge threshold. The mesh is partitioned into local mesh configurations that include the target element or edge. The operations are applied to the extracted contours of the configurations. The reconnection and size control operations use the connectivity network NN_3 to predict the connectivity of contours. The vertex repositioning and surface control operations use NN_S and NN_S^* networks to predict the coordinates of a new vertex that is of optimal position quality wise, given that the contour vertices are connected to it.

Compared with existing operations, the set of proposed operations have the potential to improve the computational cost of local improvement schemes as: (i) In comparison with the smoothing operation, the vertex repositioning and surface control operations do not involve the optimization of a local objective functional. Instead, the new optimal location of a vertex is found through the direct output of NN_S and NN_S^*. (ii) In comparison with existing topological operations that retriangulate cavities, the reconnection operation offers a computational advantage. The operation does not include the computational cost to find an optimal cavity as vertex cavitation. Moreover, unlike SPR that performs an exhaustive search for an optimal triangulation of a cavity, the reconnection operation triangulates directly the cavity using the predictions of NN_3. (iii) The size control operations for long and short edges can be viewed as composite operations that include the vertex insertion and edge contraction operation respectively. The inserted vertices regulate the edge lengths and an successive optimal triangulation for the contours that include vertices is directly predicted using NN_3.

The operations are validated and evaluated by including them in local mesh improvement schemes that are applied to static and dynamic meshes. Given a quality threshold $q_{threshold} = 0.8$, results in static meshes demonstrate the capacity of the operations to improve the quality of the perturbed meshes and obtain new meshes with a minimum quality q_{worst} close to their original state. In the worst case, a maximum $q_{worst} = 0.7$ with element angles that lie between $31° \leq \theta \leq 109°$ is reached in the example of the perturbed interface (although only the surface control operation is applied) (Fig. 5.3). When the local mesh improvement scheme (Alg. 8) is applied to dynamic meshes with a quality threshold $q_{thresh} = 0.8$, the minimum q_{worst} is improved at each simulation step in all cases. After the application of the scheme, a minimum $q_{worst} = 0.68$ is observed for the rotation of the zalesak disc (Fig. 5.9). For the same example, after the application of the scheme for each simulation step, the minimum quality lies between $0.68 \leq q_{worst} \leq 0.74$, the minimum angles lie between $35 \leq \theta_{min} \leq 44$ and the maximum angles lie between $90 \leq \theta_{max} \leq 108$. Such results conclude that the operations have the potential to be used for mesh improvement purposes. The local improvement scheme that was applied for the dynamic meshes is chosen after demonstrating the capacity to attain q_{worst} values close to the quality threshold. It is worth investigating the use of alternative local improvement schemes that use a different set of conditions and different order of operations. Moreover, since the mesh improvement speed is dependent on the selection of the quality threshold value, further investigation should involve finding a threshold-speed balance that provide a good simulation accuracy.

6 Meshing large meshes

6.1 Scheme for large mesh generation with uniform element size

The proposed neural network meshing algorithm is used to mesh larger meshes as follows. Given a high resolution contour S, vertices are sampled to form an initial contour (Fig. 6.1a) (Alg. 9, Lines 7-8). The number of vertices sampled to form the initial contour is $4 < N_C < 16$ in accordance with the trained NNs of the meshing scheme. Then, NN_3 is called to assemble the connections and output an initial mesh (Fig. 6.1b) (Alg. 9, Line 9). Assuming a target edge length l_s, for an edge e with edge length $l_e > l_s$, based on the integer ratio $K = \lfloor l_e / l_s \rfloor$, $n_K = K - 1$ equidistant vertices are inserted to e (Fig. 6.1c) (Alg .9, Line 13-21). If the vertices of an edge belong to the high resolution contour, the inserted vertices are projected to it (Fig. 6.1d) (Alg. 9, Lines 22-24). Similarly, to be in accordance with the trained NNs, the maximum number of inserted vertices per edges is $n_K = 4$. The insertion of the vertices in the elements of the initial mesh forms sub contours that are meshed using the proposed meshing scheme for the target edge length that corresponds to the average length of the edges of a sub-contour $l_s^* = \sum_{k=1}^{N} l_{e^{(k)}} / N$ (Fig. 6.1e) (Alg. 9, Line 29). Subsequently, NN_S and NN_S^* are used for the computation of mesh coordinates to improve the quality (Alg. 9, Line 31). The aforementioned refinement process is repeated until a target edge length close to l_s is reached and no further vertices are inserted (Fig. 6.1f) (Lines 27,35).

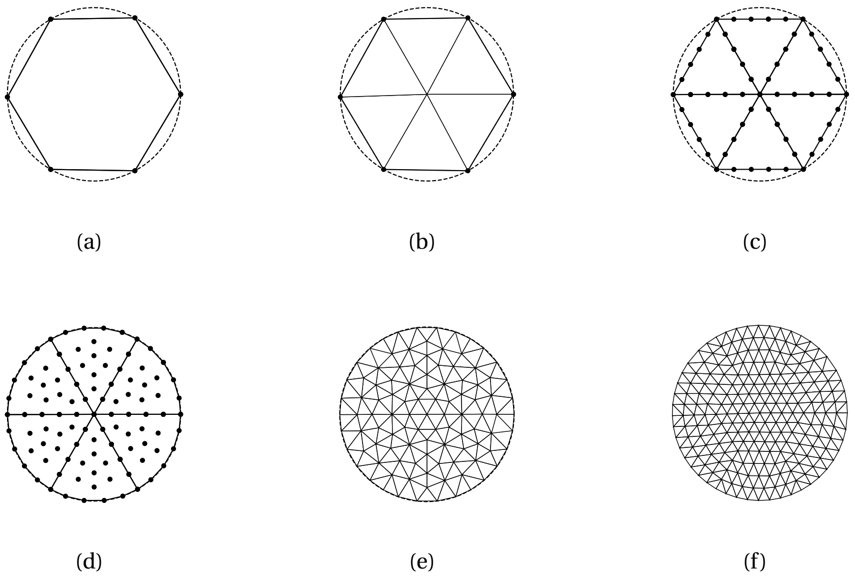


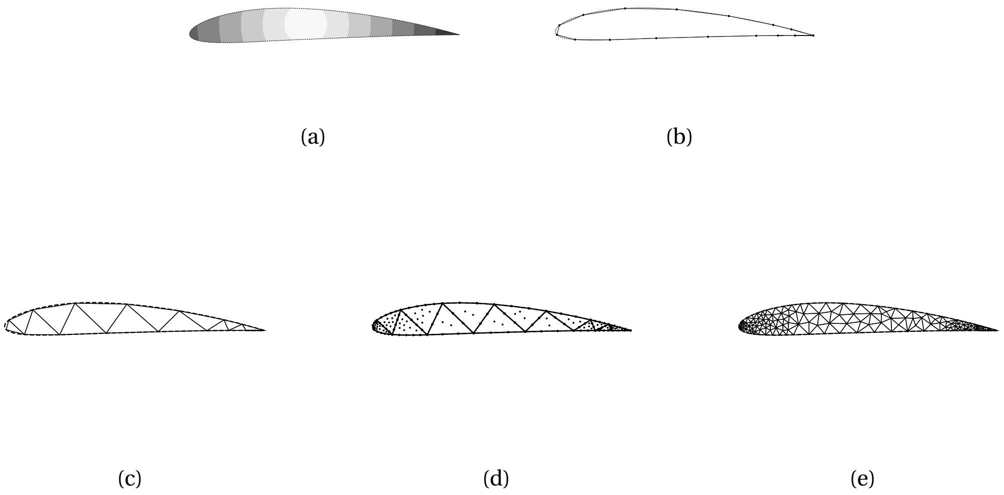
Figure 6.1: Example of the refinement process using NN_3 to create a mesh with a target edge length l_s for a high resolution contour S (150 edges) forming a circle. (a) Points are sampled from S to form an initial contour. (b) NN_3 is called to mesh the initial contour. (c) If an edge of the initial mesh has a length l_e that is bigger than l_s, then $n_K = K-1$ vertices are inserted to the edge, where $K = [l_e/l_s]$. (d) If the vertices of an edge belong to the high resolution contour S, the inserted vertices are projected to S. NN_1 and NN_2 are used to predict the number and location of inner vertices for each sub-contour with a target edge length equal to the average edge length of each sub-contour. (e) Each sub-contour composed of vertices of the elements, the inserted vertices, and the predicted inner points is meshed using NN_3. As a post-treatment, the vertex repositioning operations are applied by calling NN_S and NN_S^*. (f) The refinement process is repeated until the edge lengths are close to l_s.

Algorithm 9: Uniform scale element mesh generation using NN_3

```
$S$: high resolution contour
$l_s$: target edge length
$l_e$: length of edge e
$(p_o, p_K)$: endpoint vertices of edge e
$P$: list of inserted vertices
Sample $N_C$ vertices from $S$
Connect the $N_C$ vertices with a line to form a contour with $N_C$ edges
Form an initial mesh using $NN_3$
Refine=True
while Refine do
    foreach edge $e = (e^{(1)}, ..., e^{(i)}, .., e^{(n)})$ in current mesh $M$ do
        if $\left|l_{e^{(i)}} - l_s\right| < \epsilon$ then
            $K = [l_{e^{(i)}} / l_s]$
            if $K > 5$ then
                K=5
            end
            for j=1,2,..K-1 do
                Insert $p_j^{(i)} = p_0^{(i)} + (j/K)(p_0^{(i)} - p_K^{(i)})$
                $P \leftarrow p_j^{(i)}$
            end
            if endpoint vertices $(p_0^{(i)}, p_K^{(i)})$ of edge $e^{(i)}$ belong to the high resolution contour $S$ then
                project inserted vertices $p = (p_1^{(i)}, .., p_{K-1}^{(i)})$ to the high resolution contour $S$
            end
        end
    end
    if $P$ is not empty then
        foreach subcontour formed by the vertices of an existing element of $M$ with the inserted points $P$ do
            Mesh subcontour using the meshing scheme for a target edge length equal to the average edge length of the sub-contour
        end
        Update mesh $M$
        Apply vertex repositioning using $NN_S$ and surface control $NN_S^*$ to improve quality of $M$
        empty $P$
    else
        Refine=False
    end
end
```

6.2 Scheme for large mesh generation with adaptive element size

Adaptive mesh generation is achieved by defining a sizing function $h(\Omega_S)$ (Fig. 6.2a, Fig. 6.3a) over the inner domain of the high resolution contour Ω_S. The values of the sizing function correspond to a target edge length which may vary over the domain Ω_S; the generated mesh using a sizing function contains smaller scale element to approximate better characteristics of the geometry (e.g curvature) and larger scale elements elsewhere. In the context of the present meshing scheme, an adaptive strategy is applied by inserting vertices to the edges of an initial mesh that is generate using the connectivity network NN_3 (Fig. 6.2c, Fig. 6.3c) according to the prescribed values of the sizing function. During the refinement process, equidistant vertices $p = (p_1, .., p_{n_K})$, $n_K \leq 4$, are inserted to an edge e incrementally while $\left| l_{e^{(j)}} - h(p_j) \right| < \epsilon$ for $j = 1, 2, .., n_K$, where $l_{e^{(j)}}$ is the length of the edge $e^{(j)}$ formed by the vertices (p_j, p_{j+1}) (Alg. 10, Lines 12-26) and ϵ is a small positive number; the segment lengths of the subdivided edge should be close to the value of the sizing function of each inserted vertex. The strategy to mesh larger scale meshes with an adaptive or uniform element size inherits the automation of the meshing scheme as it is based exclusively on the use of NNs. Table 2 lists the qualities of examples used to create uniform (Fig. 6.1) and adaptive (Fig. 6.2, Fig. 6.3) large meshes.



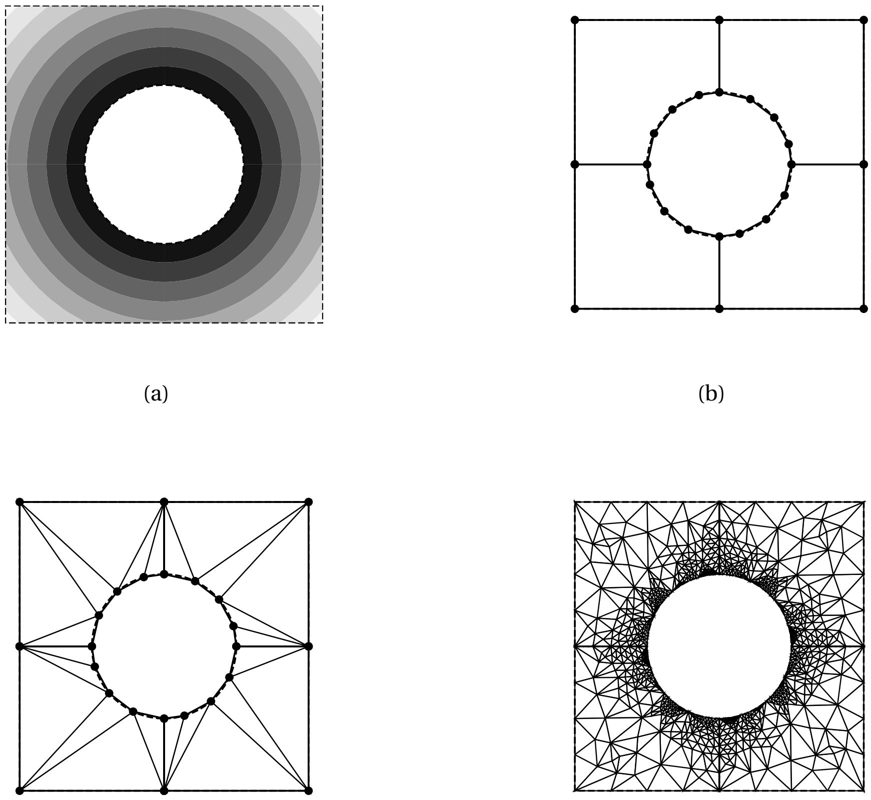
(a) (b)

(c) (d) (e)

Figure 6.2: Example of adaptive meshing for a high resolution contour S (201 edges) forming an airfoil. (a) A sizing function h is defined over the inner domain Ω_S. The values of the sizing function represent the local target edge lengths that will dictate the elements sizes. The darker areas represent smaller values of the sizing function, i.e regions where smaller elements should be created to better approximate the geometry of the airfoil. (b) Points are sampled from S to form an initial contour that is meshed using NN_3 (c). Points are inserted incrementally on each edge until all lengths of the segments that are created after the subdivision are close to the assigned size function value for each inserted vertex. The number of interior points and their location are predicted using NN_1 and NN_2. (e) Finally, after meshing each sub-contour with its inner vertices using NN_3, NN_S and NN_S^* are called to improve the quality.

Algorithm 10: Adaptive scale element mesh generation using NN_3

```
$S$: high resolution contour
$h(\Omega_S)$: sizing function defined over the domain $\Omega_S$ of the high resolution contour
$l_s$: target edge length
$l_e$: length of edge e
$\epsilon$: small positive number
$(p_o, p_K)$: endpoint vertices of edge e
$P$: list of inserted vertices
Sample $N_C$ vertices from $S$
Connect the $N_C$ vertices with a line to form a contour with $N_C$ edges
Form an initial mesh using $NN_3$
Refine=True
while Refine do
    foreach edge $e = (e^{(1)}, ..., e^{(i)}, .., e^{(n)})$ in current mesh $M$ do
        if $P$ is not empty then
            empty P
        end
        $K = 1$
        do
            K+=1
            for j=1,2,..K-1 do
                Insert $p_j^{(i)} = p_0^{(i)} + (j/K)(p_0^{(i)} - p_K^{(i)})$
                $P \leftarrow p_j^{(i)}$
            end
        while $\left| l_{e^{(j)}} - h(p_j^{(i)}) \right| < \epsilon$, where $j = 1, 2, ..K-1$ and $e^{(j)}$ is the edge with vertices
          $(p_j^{(i)}), p_{j+1}^{(i)}$, or K!=5;
        if endpoint vertices $(p_0^{(i)}, p_K^{(i)})$ of edge $e^{(i)}$ belong to the high resolution contour $S$
          then
            project inserted vertices $p = (p_1^{(i)}, .., p_{K-1}^{(i)})$ to the high resolution contour $S$
        end
    end
    if $P$ is not empty then
        foreach subcontour formed by the vertices of an existing element of M with the
          inserted points P do
            Mesh subcontour using the meshing scheme for a target edge length equal
              to the average edge length of the sub-contour
        end
        Update mesh $M$
        Apply vertex repositioning using $NN_S$ and surface control $NN_S^*$ to improve
          quality of $M$
        empty $P$
    else
        Refine=False
    end
end
```



(a)

(b)

(c)

(d)

Figure 6.3: Example of adaptive meshing for a high resolution contour with a circular hole in the middle.(a) The sizing function is defined such that elements of smaller size are created near the circular hole . (b) Since NN_3 is able to only mesh contours that are watertight (i.e no holes), the high resolution contour is divided into four sub-regions (contours containing 150 edges). Points are sampled from the sub-regions to form four contours. (c) Each of the four contours, is meshed using NN_3. (d) Based on the initial meshes, the adaptive meshing process is applied and iterated until the edge lengths of elements are close to the values of the assigned sizing function.

Example	N_{el}	θ_{min}	θ_{max}	q_{worst}	q_{mean}
Circle	384	40°	87°	0.88	0.97
Airfoil	247	31°	109°	0.69	0.90
Contour with hole	1658 (per sub-region)	28°	111°	0.66	0.85

Table 6.1: The number of elements N_{el}, minimum angles θ_{min}, maximum angles θ_{max}, worst quality q_{worst} and the mean quality q_{mean} for the circle (Fig. 6.1), airfoil (Fig. 6.2) and contour with circular hole (Fig. 6.3) examples (the closer a quality value is to 1 the better).

7 Conclusion and outlook

7.1 Conclusion

Mesh generation and improvement algorithms play an essential role in computational modeling and industrial design. Despite the advancements made, a critical issue remains the finding of a balance between automation, complexity and computational cost. Motivated by the achievements of machine learning tools in solving complex problems, this research work studies the integration of NNs to mesh generation and improvement for the development of automatic, robust, and computationally efficient frameworks.

As an initial approach to this direction, a simplicial meshing scheme for small contours is presented (**chapter 2**). Given a target edge length, the scheme is based on the use of three NNs : (i) a NN that predicts the number of inserted vertices inside the cavity of a contour, (ii) a NN that predicts the location of the inner vertices, and (iii) a NN that predicts the connectivity. The meshing scheme is trained using meshed contour datasets and generates meshes without any post treatment by providing information about a contour and a target element size. The resulting mesh is guaranteed not to contain any intersecting elements using an intermediate meshing algorithm that is based on predictions of the NN. The meshing algorithm uses a connection table to form elements by connecting facets of a contour with vertices. The formation invalid elements is avoided as: (i) The entries of the connection table that represent the connection of facets with vertices to form elements outside the domain of a contour are omitted to zero (ii) A locking mechanism excludes vertices and facets from further connections that lead to intersecting elements (**Appendix A.3.1**) (iii) A subroutine to spot sub-contours formed during the triangulation avoids connections that cross existing mesh elements (**Appendix A.3.2**). Since the intermediate meshing algorithm used in the meshing scheme is not coupled with the underlying reference mesher, this offers the possibility to adapt the scheme to the behavior of any 2D simplicial meshing algorithm. The meshing scheme offers also a computational advantage over previous works using machine learning techniques, since the number and location of inner vertices that are inserted in the cavity of the contour to satisfy element size criteria is predicted a priori the connection phase. This offers a more direct pathway to acquire a mesh that satisfies element size criteria; it avoids the incremental creation of a mesh by inserting inner vertices or elements one by one and skips intermediate topological consistency (manifold) checks.

The accuracy of the scheme is evaluated by comparing the quality of the mesh generated by the neural networks with that generated by a reference mesher that applies Constrained Delaunay Triangulation (CDT) (**chapter 3**). Based on an element quality metric, after conducting tests on contours for $N_C = \{4, 8, 10, 12, 14, 16\}$ number of edges, the results show a maximum average deviation of 27.3 % on the minimum quality between the elements of the meshes generated by the scheme and the ones generated from the reference mesher; this level of error corresponds to variation in element angles between $28° \leq \theta \leq 106°$ in the worst case. Therefore, the scheme is able to produce good quality meshes that are suitable for meshing purposes. To attain the demonstrated results, the training contour populations are increased in an exponential rate with the number of edges. Moreover, to efficiently predict the connectivity of the contours an augmentation scheme is applied; for predicting the connectivity of a contour with N_I inner vertices, multiple groups of N_I inner vertices are sampled randomly using a target edge length criteria to be included in the training dataset. The number of groups increases with the number of N_I vertices. This data augmentation process leads to the accumulation of large training datasets. To reduce the amount of training data using this process, an adaptive strategy is studied; it is shown that using the adaptive sampling strategy for a group of contours with $N_C = 10$ edges with $N_I = 4$ vertices, a 27 % higher accuracy can be achieved with a 17 % less sample population compared to the random sampling of inner vertices.

The trained NNs of the meshing scheme are used for the development of local mesh improvement operations (**chapter 4**). The operations improve the quality of elements that are below a quality threshold and regulate edge lengths according to short and long edge length thresholds. The connectivity networks are used for the development of the reconnection operation. The reconnection operation retriangulates contours of local mesh configurations that contain bad quality elements. The connectivity networks are also used to develop size control operations. The size control operation inserts vertices in the middle of target edges and retriangulates the contour of local mesh configurations that includes the edges. Two new set of NNs are also introduced for the development of vertex repositioning and surface control operations. The vertex reposition and surface control operations are based on the prediction of the newly introduced NNs that output the coordinates of a vertex. The location of the vertex is optimal quality-wise, given that the edges of the contour that includes the vertex are connected with it to form elements. When included in local mesh improvement schemes, the operations have the potential to improve the computational cost. Compared with local smoothing that moves vertices of a mesh according to the optimization of a local objective functional, the new positions of the vertices using the vertex repositioning and surface control operations are found using the prediction of NNs. Compared with vertex cavitation and the SPR operations that retriangulate cavities including bad quality elements, the reconnection operation does not entail the computational cost of finding an optimal cavity or performing an exhaustive search to find an optimal triangulation of the cavity.

The operations are validated and evaluated by including them in local mesh improvement schemes (**chapter 5**). The schemes are used to improve the quality of static and dynamic meshes. The static meshes have their vertices perturbed and edges randomly flipped to produce bad quality elements. By applying the local improvement scheme to the perturbed meshes, the quality of all test cases is improved. In the worst case, after the application of the scheme, the mesh includes element angles that lie between $31° \leq \theta \leq 109°$. By applying a local mesh improvement scheme to the dynamic meshes, the operations are able to improve the worst quality at each simulation step for all test cases. In the worst case, the application of the local mesh improvement scheme results in element minimum angles that lie between $35° \leq \theta_{min} \leq 44°$ and maximum angles that lie between $90° \leq \theta_{max} \leq 108°$ during the course of the simulation. The results confirm that the operations can be used for the mesh improvement purposes.

Finally, all the trained NNs are used to develop an iterative machine learning meshing scheme for the creation of uniform and adaptive meshes (**chapter 6**). Based on a high resolution contour, vertices are sampled to create a low resolution contour. The connectivity networks of the meshing scheme for small contours is called to generate an initial mesh. Next, vertices are inserted to the edges of the initial mesh. The number of inserted vertices is dependent on the desirable target edge length and the maximum number of contour edges the NNs of the meshing scheme are trained for. The inserted vertices are then projected to the high resolution contour. Vertices are inserted in the interior of each sub-contour using the NNs of the meshing scheme for small contours. The connectivity networks are then used to mesh each sub-contour. Finally, the vertex repositioning and surface control operations are applied to improve the quality of the resulted mesh. The aforementioned process is repeated until a target edge length is met. Examples demonstrate that the iterative algorithm generates meshes containing up to $1,658$ elements with angles that lie between $28° \leq \theta \leq 111°$.

The meshing scheme for small contours overcomes some limitations of previous approaches using NNs with supervised learning for mesh generation (Yao et al. (2005), Vinyals et al. (2015), Zhang et al. (2020)) and is more automatic as it is able to: (i) generate topologically valid meshes and provide full triangle coverage of a domain (ii) generate meshes without the use of an external meshing algorithm (iii) be trained using an automatic procedure without the need of manual exploration for training patterns. Furthermore, for the generation of large meshes, the extended scheme offers a more direct approach for mesh generation when compared to unsupervised learning NNs; the use of SOM, LIG and GNG entail a computational complexity (the search for BMU) and many iterations to converge to an acceptable mesh. Finally, to the author's knowledge, the present research work includes a first study for the development of local improvement operations using supervised learning NNs.

7.2 Outlook

The presented work leaves room for further investigation and improvements. Suggestions for new research possibilities are discussed below.

Reduction of training data size

The contour data of N_C edges used for the training of the NNs involves the random selection of N_C points from N_C divided sectors of a unitary circle. To avoid the redundancy of the training data population by generating similar contours, a dimensionality reduction algorithm could be applied. The algorithm could project the contour population into a lower dimension space (e.g 2D) where every point represents a contour. Therefore, by employing a sampling strategy of points from the low dimension space (e.g uniform sampling), a reduced training population can be created that avoids the inclusion of similar contours. Moreover, such a projection into a lower dimension space provides a clearer overview of possible "states" in the higher dimension contour space. This allows for an exploration in the contour space with the use of reinforcement learning for the NNs involved. By sampling points from the lower dimension space, the NNs could learn through a trial and error procedure where good outcomes are rewarded and bad outcomes are penalized.

The training of the connectivity network is based on an augmentation scheme. For a contour, multiple inner vertices have to be sampled to efficiently regress the values of the connection table. This augmentation in the training population is necessary as there is no permutation invariant way of ordering the inner vertices to be given as an input to the NNs. An adaptive strategy to sample the inner vertices was shown to be efficient to reduce the population of inner vertices to be sampled and therefore the training population size. However, after a certain number of inner vertices ($N_I > 4$) the strategy can be computationally expensive. Therefore, further investigation is needed for the reduction of the connectivity network's training population; strategies to this direction could either involve the development of a more sophisticated sampling strategy or the implementation of a NN architecture that is invariant to the order of input and respects to the structure of the connection table.

NN hyperparameters and other regression models

The presented hyperparameters of the NNs involved in the development of the mesh generation scheme and the creation of mesh improvement operations rely on an optimal choice after multiple trial and error tests. For a more efficient tuning of the NNs architecture, it is worth investigating the development of NNs based on a evolutionary algorithm that will be able to provide a clearer perspective regarding the choice of hyperparameters. It is also worth investigating the use of other machine learning non-linear regression models (e.g Random forests) for their adaption to the scheme.

Extension to 3D

The meshing scheme for small contours is transparent to 3D for tetrahedral mesh generation. Given a contour composed of triangular faces, the step of predicting the number of inner vertices is straight forward while the prediction of their location is achievable with the use of a 3D grid. Since the formation of elements using the meshing scheme relies on the connection of facet with a vertex to create element, the scheme could be extended to create tetrahedral elements by connecting triangular faces with a vertex using a connection table. The implementation in 3D needs to take the following into account:

- **Vertex ordering**: The ordering of the vertices for 2D contour is anti-clock wise to underlie the connections of the edges and apply the Procrustes superposition to a reference contour with a similar ordering. To underlie the connection of faces in 3D, an ordering for the vertices of the 3D contour could be based on a depth first search (DFS) vertex visit.
- **Tetrahedralization algorithm**: The triangulation algorithm of the meshing scheme visits edges and connects them with a vertex according to an ordered connection table. The connection table is ordered based on the highest entry values of each edge. In 3D, however, such an ordering could lead to the formation of self-intersecting elements. To avoid this, a tetrahedralization algorithm could be based on visiting triangular faces with high entry values followed by their neighbor faces. If any sub-contours are formed visiting all the faces, the tetrahedralization algorithm could be called recursively to mesh them.
- **Training data size**: The extension to a higher dimension increases the complexity of good quality mesh generation. Therefore, for efficient predictions of the NNs involved, a higher amount of training data will be needed compared to the 2D case. An initial population of 3D contours was generated to study the extension of the meshing scheme for small contours in 3D. A large portion of computational time involves the calculation of the connection table for the 3D contours. As an initial goal population, a number of contours was chosen that increases with the number of faces. Starting from 80000 contours with 12 faces, this number increases with the number of faces. The increase in the number, at a first stage, was not exponential as in the 2D case. This initial goal population would give an starting perspective for the behaviour of the NNs involved in the meshing scheme. An estimation on the amount of time needed to reach this initial goal population is provided in Table 7.1. The training data are generated in a machine with 64 GB memory and 2 CPUs Intel© Xeon© E5-2660v2 running at 2.2 GHz and 10 cores and a machine with 128GB memory with the same 2 CPUs.

No. faces	Goal population	Weekly growth rate(%)
12	80000	30.1
16	160000	13.57
32	640000	5.8
64	12800000	1.2
128	24000000	0.2

Table 7.1: Weekly growth rate (%) of 3D contour training datasets along with the intended initial goal population.

Furthermore, the developed operations could be extended for improving surface meshes that represent the boundary geometry of 3-D objects with 2-D simplicial elements. In this case, additional restrictions must be taken to respect the underlying surface of the mesh. The developed surface control operation predicts the optimal location of a vertex that lies in a curve. The operation could be extended such that it predicts the optimal location of a vertex that lies in a 2-D surface. In this case, the optimal position of the vertex is calculated using quadric smoothing that minimizes the distance of the vertex to planes created by the neighbor element faces contained in the contour of the local mesh configuration. Information of the vertices optimal location, the planes created by the neighbor elements faces, and the coordinates of the contour that include the vertex could be included in a dataset that trains a NN to perform the operation. The operations that change the connectivity of the mesh (reconnection, size control) could be applied to projected contours of the local mesh configurations in the 2-D plane. The connectivity information of the operation's outcome could then be mapped back in the 3-D space.

Code optimization and parallelization

All the developed algorithms are written in *Python*. Tests indicate that the perfomance of the meshing scheme for small contours is approximately four times slower than the reference mesher which is written in *C++*. The majority of mesh generation and improvement algorithms are written in low level programming languages (i.e *C++*). Therefore, for a proper study in computational time comparison, the algorithms should be transported in a low level language where they could be optimized for a maximum performance.

Moreover, apart from the meshing scheme for small contours, the algorithms of mesh improvement operation and large mesh generation are also transportable to acceleration platforms such as GPU and FPGA architectures. The local mesh improvement operations could be parallelized as each operation is applied to contours that do not share any connections. As such, each contour can be processed independently. Similarly, the meshing scheme for large meshes involves the application of the meshing of multiple small sub-contours. This allows for the parallel application of the algorithm by meshing independently each of the sub-contours.

Bibliography

Ahn Chang-Hoi, Lee Sang-Soo, Lee Hyuek-Jae, & Lee Soo-Young (1991). A self-organizing neural network approach for automatic mesh generation. *IEEE Transactions on Magnetics*, 27(5), 4201–4204.

Alfonzetti, S., Coco, S., Cavalieri, S., & Malgeri, M. (1996). Automatic mesh generation by the let-it-grow neural network. *IEEE Transactions on Magnetics*, 32(3), 1349–1352.

Alliez, P., Cohen-Steiner, D., Yvinec, M., & Desbrun, M. (2005). Variational Tetrahedral Meshing. *ACM Trans. Graph.*, 24(3), 617–625.

Allwright, S. E. (1988). *Techniques in Multiblock domain decomposition and surface grid generation*, (pp. 559–568). Pineridge Press.

Baker, T. J. (1997). Mesh adaptation strategies for problems in fluid dynamics. *Finite Elements in Analysis and Design*, 25(3), 243 – 273. Adaptive Meshing, Part 2.

Baqué, P., Remelli, E., Fleuret, F., & Fua, P. (2018). Geodesic Convolutional Shape Optimization. *CoRR*, abs/1802.04016.

Bargteil, A. W., Wojtan, C., Hodgins, J. K., & Turk, G. (2007). A Finite Element Method for Animating Large Viscoplastic Flow. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07 New York, NY, USA: ACM.

Beniere, R., Subsol, G., Gesquière, G., Breton, F. L., & Puech, W. (2013). A comprehensive process of reverse engineering from 3D meshes to CAD models. *Computer-Aided Design*, 45(11), 1382 – 1393.

Boscaini, D., Masci, J., Rodolà, E., & Bronstein, M. (2016). Learning shape correspondence with anisotropic convolutional neural networks. In *Advances in neural information processing systems* (pp. 3189–3197).

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., & Vandergheynst, P. (2017). Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34(4), 18–42.

Bruna, J., Zaremba, W., Szlam, A., & Lecun, Y. (2014). Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations (ICLR2014), CBLS, April 2014*.

Caendish, J. C., Field, D. A., & Frey, W. H. (1985). An apporach to automatic three-dimensional finite element mesh generation. *International journal for numerical methods in engineering*, 21(2), 329–347.

Calisto, M. B. & Lai-Yuen, S. K. (2020). Adaen-net: An ensemble of adaptive 2D-3D Fully Convolutional Networks for medical image segmentation. *Neural Networks*, 126, 76 – 94.

Chen, J., Zheng, J., Zheng, Y., Xiao, Z., Si, H., & Yao, Y. (2017). Tetrahedral mesh improvement by shell transformation. *Eng. with Comput.*, 33(3), 393–414.

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. (2016). Deeplab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PP.

Chen, X. & Yang, V. (2014). Thickness-based adaptive mesh refinement methods for multiphase flow simulations with thin regions. *Journal of Computational Physics*, 269(Supplement C), 22 – 39.

Cheng, S.-W., Dey, T. K., Edelsbrunner, H., Facello, M. A., & Teng, S.-H. (2000). Silver Exudation. *J. ACM*, 47(5), 883–904.

Cheng, S.-W., Dey, T. K., & Shewchuk, J. (2012). *Delaunay mesh generation*. CRC Press.

Chew, L. P. (1989). *Guaranteed-quality triangular meshes*. Technical report, CORNELL UNIV ITHACA NY DEPT OF COMPUTER SCIENCE.

Clausen, P., Wicke, M., Shewchuk, J. R., & O'Brien, J. F. (2013). Simulating Liquids and Solid-liquid Interactions with Lagrangian Meshes. *ACM Trans. Graph.*, 32(2), 17:1–17:15.

Cook, W. A. (1974). Body oriented (natural) co-ordinates for generating three-dimensional meshes. *International Journal for Numerical Methods in Engineering*, 8(1), 27–43.

Dassi, F., Kamenski, L., Farrell, P., & Si, H. (2018). Tetrahedral mesh improvement using moving mesh smoothing, lazy searching flips, and rbf surface reconstruction. *Computer-Aided Design*, 103, 2 – 13. 25th International Meshing Roundtable Special Issue: Advances in Mesh Generation.

de L'isle, E. B. & George, P. L. (1995). Optimization of tetrahedral meshes. In I. Babuska, W. D. Henshaw, J. E. Oliger, J. E. Flaherty, J. E. Hopcroft, & T. Tezduyar (Eds.), *Modeling, Mesh Generation, and Adaptive Numerical Methods for Partial Differential Equations* (pp. 97–127). New York, NY: Springer New York.

Defferrard, M., Bresson, X., & Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems* (pp. 3844–3852).

Donea, J., Giuliani, S., & Halleux, J. (1982). An arbitrary lagrangian-eulerian finite element method for transient dynamic fluid-structure interactions. *Computer Methods in Applied Mechanics and Engineering*, 33(1), 689 – 723.

Dvinsky, A. S. (1991). Adaptive grid generation from harmonic maps on riemannian manifolds. *Journal of Computational Physics*, 95(2), 450–476.

Edelsbrunner, H., Li, X.-Y., Miller, G., Stathopoulos, A., Talmor, D., Teng, S.-H., Üngör, A., & Walkington, N. (2000). Smoothing and cleaning up slivers. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00 (pp. 273–277). New York, NY, USA: Association for Computing Machinery.

Egmont-Petersen, M., de Ridder, D., & Handels, H. (2002). Image processing with neural networks—a review. *Pattern recognition*, 35(10), 2279–2301.

ElAdel, A., Zaied, M., & Amar, C. B. (2017). Fast DCNN based on FWT, intelligent dropout and layer skipping for image retrieval. *Neural Networks*, 95, 10 – 18.

Fischer, A. & Bar-Yoseph, Z. (2000). Adaptive mesh generation based on multiresolution quadtree representation. *International Journal for Numerical Methods in Engineering*, 48, 1571 – 1582.

Fort, J. (2006). SOM's mathematics. *Neural Networks*, 19(6), 812 – 816. Advances in Self Organising Maps - WSOM'05.

Freitag, L., Jones, M., & Plassmann, P. (1995). An efficient parallel algorithm for mesh smoothing. In *INTERNATIONAL MESHING ROUNDTABLE* (pp. 47–58).

Freitag, L. A. & Ollivier-Gooch, C. (1997a). Tetrahedral mesh improvement using swapping and smoothing. *International Journal for Numerical Methods in Engineering*, 40(21), 3979–4002.

Freitag, L. A. & Ollivier-Gooch, C. (1997b). Tetrahedral mesh improvement using swapping and smoothing. *International Journal for Numerical Methods in Engineering*, 40(21), 3979–4002.

Frey, P. J. & George, P.-L. (2007). *Mesh Generation: Application to Finite Elements*. ISTE.

Fritzke, B. (1995). A growing neural gas network learns topologies. In *Advances in neural information processing systems* (pp. 625–632).

George, J. A. (1971). Computer implementation of the finite element method.

George, P., Hecht, F., & Saltel, E. (1991). Automatic mesh generator with specified boundary. *Computer Methods in Applied Mechanics and Engineering*, 92(3), 269 – 288.

Gower, J. C. (1975). Generalized procrustes analysis. *Psychometrika*, 40(1), 33–51.

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural turing machines.

Greaves, D. M. & Borthwick, A. G. L. (1999). Hierarchical tree-based finite element mesh generation. *International Journal for Numerical Methods in Engineering*, 45(4), 447–471.

Grigorescu, S., Trasnea, B., Cocias, T., & Macesanu, G. (2020). A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3), 362–386.

Guennebaud, G. & Gross, M. (2007). Algebraic Point Set Surfaces. *ACM Trans. Graph.*, 26(3).

Guo, B., Wang, J., Jiang, X., Li, C., Su, B., Cui, Z., Sun, Y., & Yang, C. (2020). A 3d surface reconstruction method for large-scale point cloud data. *Mathematical Problems in Engineering*, 2020, 1–14.

Guo, J., Ding, F., Jia, X., & Yan, D.-M. (2019). Automatic and high-quality surface mesh generation for CAD models. *Computer-Aided Design*, 109, 49 – 59.

Hanocka, R., Hertz, A., Fish, N., Giryes, R., Fleishman, S., & Cohen-Or, D. (2019). MeshCNN: a network with an edge. *ACM Transactions on Graphics*, 38, 1–12.

Hassan, O., Morgan, K., Probert, E., & Peraire, J. (1996). Unstructured tetrahedral mesh generation for three-dimensional viscous flows. *International Journal for Numerical Methods in Engineering*, 39(4), 549–567.

He, L., Zheng, J., Zheng, Y., Chen, J., Zhou, X., & Xiao, Z. (2019). Parallel algorithms for moving boundary problems by local remeshing. *Engineering Computations*.

Henaff, M., Bruna, J., & LeCun, Y. (2015). Deep Convolutional Networks on Graph-Structured Data. *CoRR*, abs/1506.05163.

Hermeline, F. (1982). Triangulation automatique d'un polyèdre en dimension n. *ESAIM: Mathematical Modelling and Numerical Analysis - Modélisation Mathématique et Analyse Numérique*, 16(3), 211–242.

Holdstein, Y. & Fischer, A. (2008). Three-dimensional surface reconstruction using meshing growing neural gas (mgng). *The Visual Computer*, 24(4), 295–302.

Hu, Y., Zhou, Q., Gao, X., Jacobson, A., Zorin, D., & Panozzo, D. (2018). Tetrahedral Meshing in the Wild. *ACM Trans. Graph.*, 37(4), 60:1–60:14.

Huang, W. (2001). Variational mesh adaptation: isotropy and equidistribution. *Journal of Computational Physics*, 174(2), 903–924.

Huang, W. & Russell, R. D. (2010). *Adaptive moving mesh methods*, volume 174. Springer Science & Business Media.

Hughes, T., Cottrell, J., & Bazilevs, Y. (2005). Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. *Computer Methods in Applied Mechanics and Engineering*, 194(39), 4135 – 4195.

Kallinderis, Y., Khawaja, A., & McMorris, H. (1995). *Hybrid prismatic/tetrahedral grid generation for complex geometries.*

Kipf, T. N. & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*: OpenReview.net.

Klincsek, G. (1980). Minimal triangulations of polygonal domains. In P. L. Hammer (Ed.), *Combinatorics 79*, volume 9 of *Annals of Discrete Mathematics* (pp. 121 – 123). Elsevier.

Klingner, B. M. & Shewchuk, J. R. (2008). Aggressive tetrahedral mesh improvement. In *Proceedings of the 16th international meshing roundtable* (pp. 3–23).

Knupp, P. M. (1996). Jacobian-weighted elliptic grid generation. *SIAM Journal on Scientific Computing*, 17(6), 1475–1490.

Knupp, P. M. & Robidoux, N. (2000). A framework for variational grid generation: conditioning the jacobian matrix with matrix norms. *SIAM Journal on Scientific Computing*, 21(6), 2029–2047.

Kohonen, T. (2013). Essentials of the self-organizing map. *Neural Networks*, 37, 52 – 65. Twenty-fifth Anniversay Commemorative Issue.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 25* (pp. 1097–1105). Curran Associates, Inc.

Labelle, F. & Shewchuk, J. R. (2007). Isosurface stuffing: Fast tetrahedral meshes with good dihedral angles. *ACM Trans. Graph.*, 26(3), 57–es.

Lambrechts, J., Comblen, R., Legat, V., Geuzaine, C., & Remacle, J.-F. (2008). Multiscale mesh generation on the sphere. *Ocean Dynamics*, 58(5), 461–473.

Lavoué, G., Dupont, F., & Baskurt, A. (2005). A new CAD mesh segmentation method, based on curvature tensor analysis. *Computer-Aided Design*, 37(10), 975 – 987.

LeCun, Y. (2012). Learning Invariant Feature Hierarchies. In *European conference on computer vision* (pp. 496–505).

Li, X. (2001). *Sliver-Free Three Dimensional Delaunay Mesh Generation.* PhD thesis, USA. AAI9996652.

Liang, Q., Rodriguez, C., Egusquiza, E., Escaler, X., Farhat, M., & Avellan, F. (2007). Numerical simulation of fluid added mass effect on a francis turbine runner. *Computer & Fluids*, 36, 1106–1118.

Litany, O., Bronstein, A., Bronstein, M., & Makadia, A. (2018). Deformable shape completion with graph convolutional autoencoders. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1886–1895).

Liu, J., Chen, Y. Q., & Sun, S. L. (2009). Small polyhedron reconnection for mesh improvement and its implementation based on advancing front technique. *International Journal for Numerical Methods in Engineering*, 79(8), 1004–1018.

Liu, Y., Saputra, A. A., Wang, J., Tin-Loi, F., & Song, C. (2017). Automatic polyhedral mesh generation and scaled boundary finite element analysis of STL models. *Computer Methods in Applied Mechanics and Engineering*, 313, 106–132.

Lohner, R. (1995). Mesh adaptation in fluid mechanics. *Engineering Fracture Mechanics*, 50(5), 819 – 847.

Löhner, R. (1996). Extensions and improvements of the advancing front grid generation technique. *Communications in Numerical Methods in Engineering*, 12(10), 683–702.

Löhner, R. & Parikh, P. (1988). Generation of three-dimensional unstructured grids by the advancing-front method. *International Journal for Numerical Methods in Fluids*, 8(10), 1135–1149.

López-Rubio, E. & Ramos, A. D. (2014). Grid topologies for the self-organizing map. *Neural Networks*, 56, 35 – 48.

Ma, X. & Sun, L. (2019). An automatic approach to constrained quadrilateral mesh generation. *Engineering Computations*, 37, 929–951.

Malvern, L. E. (1969). *Introduction to the Mechanics of a Continuous Medium*. Number Monograph.

Manevitz, L., Yousef, M., & Givoli, D. (1997). Finite Element Mesh Generation Using Self Organizing Neural Networks. *Computer-Aided Civil and Infrastructure Engineering*, 12(4), 233–250.

Marcum, D. L. & Weatherill, N. P. (1995). Unstructured grid generation using iterative point insertion and local reconnection. *AIAA Journal*, 33(9), 1619–1625.

Maréchal, L. (2001). A new approach to octree-based hexahedral meshing. In *IMR*.

Maron, H., Galun, M., Aigerman, N., Trope, M., Dym, N., Yumer, E., Kim, V. G., & Lipman, Y. (2017). Convolutional Neural Networks on Surfaces via Seamless Toric Covers. *ACM Trans. Graph.*, 36(4).

Marot, C. & Remacle, J.-F. (2020). Quality tetrahedral mesh generation with hxt.

Martinetz, T. & Schulten, K. (1994). Topology representing networks. *Neural Networks*, 7(3), 507 – 522.

Marton, Z. C., Rusu, R. B., & Beetz, M. (2009). On fast surface reconstruction methods for large and noisy point clouds. In *2009 IEEE international conference on robotics and automation* (pp. 3218–3223).: IEEE.

Masci, J., Boscaini, D., Bronstein, M., & Vandergheynst, P. (2015). Geodesic convolutional neural networks on riemannian manifolds. In *Proceedings of the IEEE international conference on computer vision workshops* (pp. 37–45).

Mavriplis, D. J. (1995). An Advancing Front Delaunay Triangulation Algorithm Designed for Robustness. *Journal of Computational Physics*, 117(1), 90 – 101.

Melato, M., Hammer, B., & Hormann, K. (2007). Neural gas for surface reconstruction. *Institut für Informatik-IfI Technical Report Series.*

Merriam, M. (1991). *An efficient advancing front algorithm for Delaunay triangulation.*

Monti, F., Boscaini, D., Masci, J., Rodolà, E., Svoboda, J., & Bronstein, M. M. (2017). Geometric Deep Learning on Graphs and Manifolds Using Mixture Model CNNs. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 5425–5434).

Morozov, D. & Peterka, T. (2016). Efficient delaunay tessellation through k-d tree decomposition. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 728–738).

Nechaeva, O. (2006). Composite Algorithm for Adaptive Mesh Construction Based on Self-Organizing maps. In S. D. Kollias, A. Stafylopatis, W. Duch, & E. Oja (Eds.), *Artificial Neural Networks – ICANN 2006* (pp. 445–454). Berlin, Heidelberg: Springer Berlin Heidelberg.

Neil Molino, R. B. & Fedkiw, R. (2003). Tetrahedral mesh generation for deformable bodies. In *In Proc. Symposium on Computer Animation.*

Parthasarathy, V., Graichen, C., & Hathaway, A. (1994). A comparison of tetrahedron quality measures. *Finite Elements in Analysis and Design*, 15(3), 255 – 261.

Parthasarathy, V. & Kodiyalam, S. (1991). A constrained optimization approach to finite element mesh smoothing. *Finite Elements in Analysis and Design*, 9(4), 309 – 320.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., & Lerer, A. (2017). Automatic differentiation in PyTorch. In *NIPS-W*.

Paul Chew, L. (1989). Constrained Delaunay triangulations. *Algorithmica*, 4(1), 97–108.

Pochet, A., Filho, W. C., Lopes, H., & Gattass, M. (2016). A new quadtree-based approach for automatic quadrilateral mesh generation. *Engineering with Computers*, 33, 275–292.

Portaneri, C., Alliez, P., Hemmer, M., Birklein, L., & Schoemer, E. (2019). Cost-driven Framework for Progressive Compression of Textured Meshes. In *Proceedings of the 10th ACM Multimedia Systems Conference*, MMSys '19 (pp. 175–188). New York, NY, USA: ACM.

Rebay, S. (1993). Efficient Unstructured Mesh Generation by Means of Delaunay Triangulation and Bowyer-Watson Algorithm. *Journal of Computational Physics*, 106(1), 125 – 138.

Rivara, M.-C. (1997). New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. *International Journal for Numerical Methods in Engineering*, 40, 3313–3324.

Rodriguez, M. & Maria, C. (2017). Lepp-WCentroid method for tetrahedral mesh improvement.

Ruppert, J. (1993). A new and simple algorithm for quality 2-dimensional mesh generation”. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, volume 66 (pp.$\tilde{8}$3).: SIAM.

Scheiders, R. (2000). Octree-based hexahedral mesh generation. *International Journal of Computational Geometry & Applications*, 10(04), 383–398.

Schneiders, R. & Bünten, R. (1995). Automatic generation of hexahedral finite element meshes. *Computer Aided Geometric Design*, 12(7), 693 – 707. Grid Generation, Finite Elements, and Geometric Design.

Schroeder, W. J. & Shephard, M. S. (1990). A combined octree/delaunay method for fully automatic 3-d mesh generation. *International Journal for Numerical Methods in Engineering*, 29(1), 37–55.

Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., & LeCun, Y. (2014). Overfeat: Integrated Recognition, Localization and Detection using Convolutional Networks. *CoRR*, abs/1312.6229.

Seveno, E. et al. (1997). Towards an adaptive advancing front method. In *6th International Meshing Roundtable* (pp. 349–362).

Shewchuk, J. R. (2002a). Constrained Delaunay Tetrahedralizations and Provably Good Boundary Recovery. In *In Eleventh International Meshing Roundtable* (pp. 193–204).

Shewchuk, J. R. (2002b). Constrained delaunay tetrahedralizations and provably good boundary recovery. In *IMR* (pp. 193–204).: Citeseer.

Shewchuk, J. R. (2002c). What is a Good Linear Element? Interpolation, Conditioning, and Quality Measures. In *IMR*.

Si, H. & Gärtner, K. (2005). Meshing piecewise linear complexes by constrained delaunay tetrahedralizations. In B. W. Hanks (Ed.), *Proceedings of the 14th International Meshing Roundtable* (pp. 147–163). Berlin, Heidelberg: Springer Berlin Heidelberg.

Si, H. & Shewchuk, J. R. (2014). Incrementally constructing and updating constrained delaunay tetrahedralizations with finite-precision coordinates. *Engineering with Computers*, 30(2), 253–269.

Simonyan, K. & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556.

Solanpää, J. & Räsänen, E. (2018). Fiend–finite element quantum dynamics. *arXiv preprint arXiv:1812.05943*.

Sumner, R. W. & Popović, J. (2004). Deformation Transfer for Triangle Meshes. *ACM Trans. Graph.*, 23(3), 399–405.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. In *Advances in neural information processing systems* (pp. 3104–3112).

Tatarchenko, M., Park, J., Koltun, V., & Zhou, Q. (2018). Tangent Convolutions for Dense Prediction in 3D. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 3887–3896).

Thompson, J., Warsi, Z. U. A., & Mastin, C. (1985). Numerical grid generation: Foundations and applications.

Triantafyllidis, D. G. & Labridis, D. P. (2002). A finite-element mesh generator based on growing neural networks. *IEEE Transactions on Neural Networks*, 13(6), 1482–1496.

Verma, N., Boyer, E., & Verbeek, J. (2018). Feastnet: Feature-steered graph convolutions for 3D shape analysis. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2598–2606).

Vinyals, O., Fortunato, M., & Jaitly, N. (2015). Pointer Networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 28* (pp. 2692–2700). Curran Associates, Inc.

Wang, N., Zhang, Y., Li, Z., Fu, Y., Liu, W., & Jiang, Y.-G. (2018). Pixel2mesh: Generating 3D mesh models from single RGB images. In *Proceedings of the European Conference on Computer Vision (ECCV)* (pp. 52–67).

Watson, D. F. (1981). Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes*. *The Computer Journal*, 24(2), 167–172.

Weatherill, N. P. & Hassan, O. (1994). Efficient three-dimensional delaunay triangulation with automatic point creation and imposed boundary constraints. *International Journal for Numerical Methods in Engineering*, 37(12), 2005–2039.

Wen, C., Zhang, Y., Li, Z., & Fu, Y. (2019). Pixel2mesh++: Multi-view 3D mesh generation via deformation. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1042–1051).

Wicke, M., Ritchie, D., Klingner, B. M., Burke, S., Shewchuk, J. R., & O'Brien, J. F. (2010). Dynamic local remeshing for elastoplastic simulation. *ACM Transactions on graphics (TOG)*, 29(4), 1–11.

Wojtan, C. & Turk, G. (2008). Fast viscoelastic behavior with thin features. *ACM Trans. Graph.*, 27.

Yao, S., Yan, B., Chen, B., & Zeng, Y. (2005). An ANN-based element extraction method for automatic mesh generation. *Expert Syst. Appl.*, 29, 193–206.

Yerry, M. A. & Shephard, M. S. (1983). A modified quadtree approach to finite element mesh generation. *IEEE Computer Graphics and Applications*, 3(1), 39–46.

Young, T., Hazarika, D., Poria, S., & Cambria, E. (2018). Recent trends in deep learning based natural language processing. *IEEE Computational intelligenCe magazine*, 13(3), 55–75.

Zhang, Z., Wang, Y., Jimack, P. K., & Wang, H. (2020). Meshingnet: A new mesh generation method based on deep learning. In V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, & J. Teixeira (Eds.), *Computational Science – ICCS 2020* (pp. 186–198). Cham: Springer International Publishing.

Zhao, D., Chen, J., Zheng, Y., Huang, Z.-g., & Zheng, J. (2015). Fine-grained parallel algorithm for unstructured surface mesh generation. *Computer and structures*, 154, 177–191.

Zheng, J., Chen, J., Zheng, Y., Yao, Y., Li, S., & Xiao, Z. (2016). An improved local remeshing algorithm for moving boundary problems. *Engineering Applications of Computational Fluid Mechanics*, 10(1), 403–426.

A Supplementary material

A.1 Back propagation

The NN is trained by minimizing a loss function over a training dataset $D = \{(X^{(k)}, Y^{(k)})\}_{k=0}^{N}$, where $X^{(k)} = (x_1^{(k)}, .., x_n^{(k)})$ are the per sample input signals and $Y^{(k)} = (y_1^{(k)}, .., y_m^{(k)})$ are the desirable outputs. To apply gradient descent to the loss function:

$$\mathscr{L}(w, b) = \sum_{k=1}^{N} \ell(y(X^{(k)}; w, b), Y^{(k)}) \tag{A.1}$$

the expression of the gradient of the per sample loss function $\ell_k = \ell(y(X^{(k)}; w, b), Y^{(k)})$ with respect to the parameters $a = (w, b)$ must be calculated, e.g.:

$$\frac{\partial \ell_k}{\partial w_{i,j}^{[l]}} \quad \text{and} \quad \frac{\partial \ell_k}{\partial b_i^{[l]}} \tag{A.2}$$

for $l = 1, 2, ..., c + 1$.

For clarity a single input of a training example X is considered. According to the architecture of NN the input signal follows the path:

$$X^{[0]} \equiv X \xrightarrow{w^{[1]}, b^{[1]}} u^{[1]} \xrightarrow{g^{[1]}} X^{[1]} \xrightarrow{w^{[2]}, b^{[2]}} u^{[2]} \xrightarrow{g^{[2]}} X^{[2]} \xrightarrow{w^{[2]}, b^{[2]}} \ldots \xrightarrow{w^{[c]}, b^{[c]}} u^{[c]} \xrightarrow{g^{[c]}} X^{[c]} \equiv f(X; w, b) \tag{A.3}$$

where for the $L^{[l]}$ layer of the NN, $X^{[l]} = (x_1^{[l]}, ..., x_{h_l}^{[l]})$ denotes the output signal, $w^{[l]} = (w_{(1,1)}^{[l]}, ..., w_{(h_{l-1}, h_l)}^{[l]})$, $b^{[l]} = (b_1^{[l]}, ..., b_{h_l}^{[l]})$ are the set of weights and biases respectively, $u^{[l]} = w^{[l]} X^{[l-1]} + b^{[l]}$, and $g^{[l]} = g^{[l]}(u^{[l]}) \equiv X^{[l]}$ is the output of the activation functions. The input signal and output signal of the NN are formally set as $X^{[0]} \equiv X$ and $t(X; w, b) \equiv X^{[c]}$ respectively. The path

A.3 is also known as forward pass.

By examining the path between two layers:

$$X^{[l-1]} \xrightarrow{w^{[l]},b^{[l]}} u^{[l]} \xrightarrow{g^{[l]}} X^{[l]} \tag{A.4}$$

it can be observed that since $g^{[l]}$ influences the loss function ℓ of sample input X through $x_j^{[l]}$ with $x_j^{[l]} = g^{[l]}(u_j^{[l]})$, $\forall j = \{1, 2, .., h_l\}$:

$$\frac{\partial \ell}{\partial u_j^{[l]}} = \frac{\partial \ell}{\partial x_j^{[l]}} \frac{\partial x_j^{[l]}}{\partial u_j^{[l]}} = \frac{\partial \ell}{\partial x_j^{[l]}} \frac{\partial g^{[l]}}{\partial u_j^{[l]}} = \frac{\partial \ell}{\partial x_j^{[l]}} \dot{g}^{[l]}(u_j^{[l]}) \tag{A.5}$$

and since $x_i^{[l-1]}$ influences ℓ through $u_j^{[l]}$ with $u_j^{[l]} = b_j^{[l-1]} + \sum_{i=1}^{h_l} w_{ij}^{[l]} x_i^{[l-1]}$, $\forall i = \{1, 2, .., h_{l-1}\}$:

$$\frac{\partial \ell}{\partial x_i^{[l-1]}} = \sum_i \frac{\partial \ell}{\partial u_j^{[l]}} \frac{\partial u_j^{[l]}}{\partial x_i^{[l-1]}} = \sum_i \frac{\partial \ell}{\partial u_j^{[l]}} w_{i,j}^{[l]} \tag{A.6}$$

Since $u_j^{[l]} = b_j^{[l-1]} + \sum_{i=1}^{h_l} w_{ij}^{[l]} x_i^{[l-1]}$, $w_{ij}^{[l]}$ and $b_j^{[l]}$ influence ℓ through $u_j^{(l)}$, the chain rule gives:

$$\frac{\partial \ell}{\partial w_{i,j}^{[l]}} = \frac{\partial \ell}{\partial u_j^{[l]}} \frac{\partial u_j^{[l]}}{\partial w_{i,j}^{[l]}} = \frac{\partial \ell}{\partial u_j^{[l]}} x_i^{[l-1]} \tag{A.7}$$

and similarly:

$$\frac{\partial \ell}{\partial b_j^{[l]}} = \frac{\partial \ell}{\partial u_j^{[l]}} \frac{\partial u_j^{[l]}}{\partial b_j^{[l]}} = \frac{\partial \ell}{\partial u_j^{[l]}} \tag{A.8}$$

For the j neuron in the last layer $L^{[c]}$ of NN the gradient of the lost function with respect to the input is:

$$\frac{\partial \ell}{\partial x_j^{[c]}} = (\nabla \ell)_j \tag{A.9}$$

To compute the derivatives of the loss function with respect to the parameters $a = (w, b)$, $\partial \ell / \partial w_{i,j}^{[l]}$ and $\partial \ell / \partial b_j^{[l]}$, first the derivative A.9 is computed. Then by propagating backwards, the derivatives of the loss function with respect to the activation functions are computed using eq.A.5 and eq.A.6 are calculated. Finally, $\partial \ell / \partial w_{i,j}^{[l]}$ and $\partial \ell / \partial b_j^{[l]]}$ are computed using eq.A.7

and eq.A.8 respectively.

A.2 $Gmsh^{©}$ mesh generation

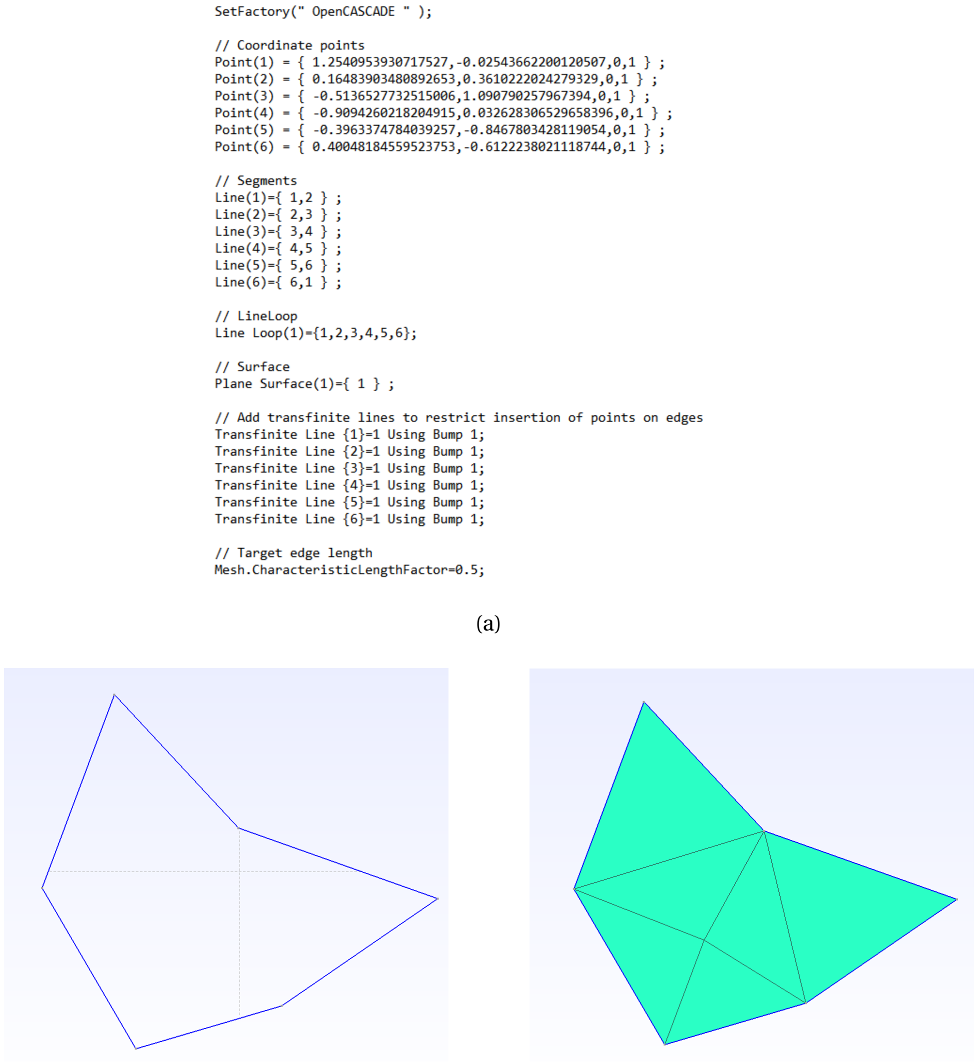
The demonstrated graded meshes for training the NNs and those that are subjected to the mesh improvement are generated using $Gmsh^{©}$. The software generates a mesh given a .geo file that describes the geometry of the input. The target edge length can be either defined uniformly or regionally for adaptive mesh generation. To achieve adaptivity in mesh element size for the demonstrated examples, the target edge length is defined by an interpolation process; the points of the geometry are assigned a desirable target edge length and the values of the target edge length for the rest of the geometry are found by interpolating the assigned values of the points. Once the input geometry ($input.geo$) and the desirable target edge length is defined, the command:

$$"input.geo\ -2\ -algo\ del2d\ output.msh"$$

is called to generate the mesh ($output.msh$). The Constrained Delaunay Triangulation ($del2d$) was used along by a refinement process to satisfy target edge length criteria (uniform or local). In what follows examples of the .geo files, the geometry and the resulted mesh are shown.

A.2.1 Contour mesh generation

The $input.geo$ file provides to the reference mesher the information on the geometry of the contour, the meshing parameters and the target edge length (Fig. A.1). The points, edges (*Segments*), contour (*LineLoop*) and cavity (*Surface*) are given as information to create the geometry of the contour. The *Trasfinite* command restricts points from being inserted at the edges of the contour. The target edge length is defined via the *Mesh.CharacteristicLengthFactor* variable.



```
SetFactory(" OpenCASCADE " );

// Coordinate points
Point(1) = { 1.2540953930717527,-0.02543662200120507,0,1 } ;
Point(2) = { 0.16483903480892653,0.3610222024279329,0,1 } ;
Point(3) = { -0.5136527732515006,1.090790257967394,0,1 } ;
Point(4) = { -0.9094260218204915,0.032628306529658396,0,1 } ;
Point(5) = { -0.3963374784039257,-0.8467803428119054,0,1 } ;
Point(6) = { 0.40048184559523753,-0.6122238021118744,0,1 } ;

// Segments
Line(1)={ 1,2 } ;
Line(2)={ 2,3 } ;
Line(3)={ 3,4 } ;
Line(4)={ 4,5 } ;
Line(5)={ 5,6 } ;
Line(6)={ 6,1 } ;

// LineLoop
Line Loop(1)={1,2,3,4,5,6};

// Surface
Plane Surface(1)={ 1 } ;

// Add transfinite lines to restrict insertion of points on edges
Transfinite Line {1}=1 Using Bump 1;
Transfinite Line {2}=1 Using Bump 1;
Transfinite Line {3}=1 Using Bump 1;
Transfinite Line {4}=1 Using Bump 1;
Transfinite Line {5}=1 Using Bump 1;
Transfinite Line {6}=1 Using Bump 1;

// Target edge length
Mesh.CharacteristicLengthFactor=0.5;
```

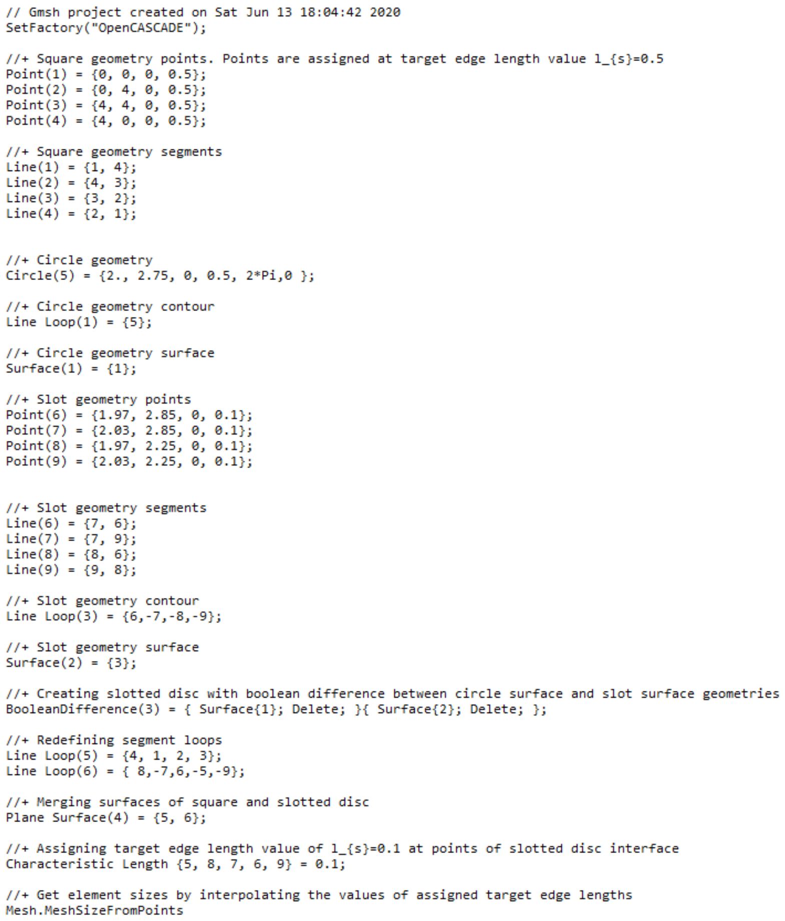
(a)

(b) (c)

Figure A.1: Example of generating a graded mesh of a contour geometry with $Gmsh^{©}$. (a) In the $input.geo$ file the points, edges (*Segments*), contour (*LineLoop*) define the geometry input. The *Trasfinite* command constraints vertices from being inserted at the edges of the contour. *Mesh.CharacteristicLengthFactor* defines the uniform target edges length. (b) The geometry of the $input.geo$ file. (c) The graded mesh after calling the command "$input.geo - 2 - algo\ del2d\ output.msh$".

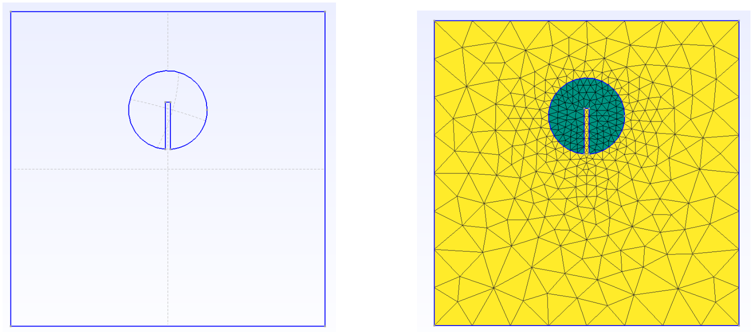
A.2.2 Test cases mesh generation

To generate meshes of adaptive size the points are assigned a target edge length value which dictates the desired element size at these points. The size of the mesh elements is then computed by interpolating these values inside the domain during mesh generation (Fig. A.2, Fig. A.3). The size is given in the form of a 4^{th} point coordinate (x, y, z, l_s), where x, y, z are the coordinates of the point and l_s is the target edge length.



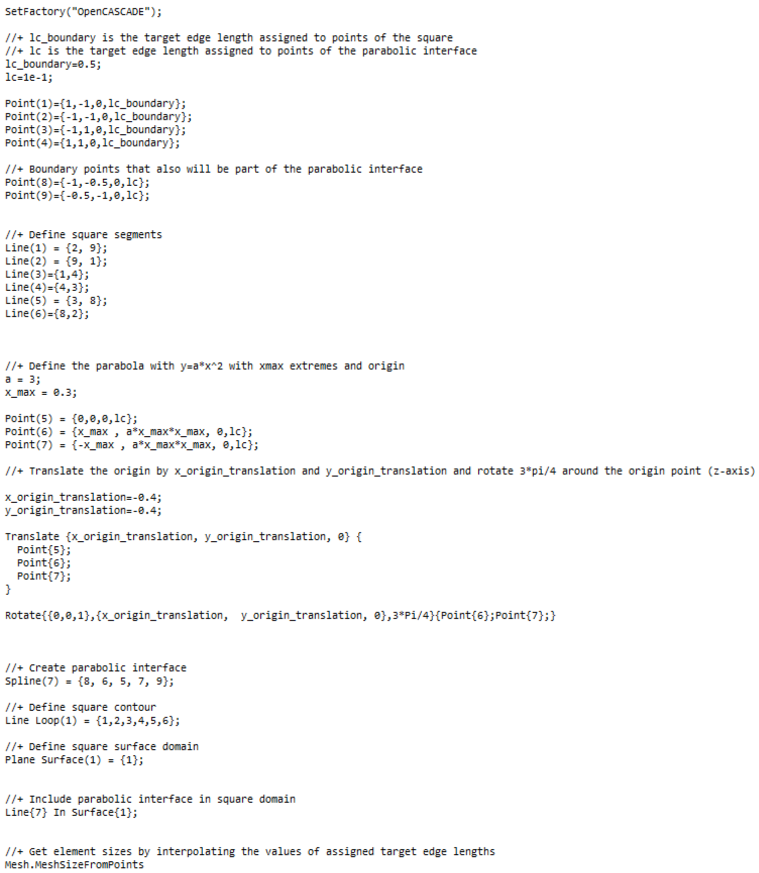
```
// Gmsh project created on Sat Jun 13 18:04:42 2020
SetFactory("OpenCASCADE");

//+ Square geometry points. Points are assigned at target edge length value l_{s}=0.5
Point(1) = {0, 0, 0, 0.5};
Point(2) = {0, 4, 0, 0.5};
Point(3) = {4, 4, 0, 0.5};
Point(4) = {4, 0, 0, 0.5};

//+ Square geometry segments
Line(1) = {1, 4};
Line(2) = {4, 3};
Line(3) = {3, 2};
Line(4) = {2, 1};


//+ Circle geometry
Circle(5) = {2., 2.75, 0, 0.5, 2*Pi,0 };

//+ Circle geometry contour
Line Loop(1) = {5};

//+ Circle geometry surface
Surface(1) = {1};

//+ Slot geometry points
Point(6) = {1.97, 2.85, 0, 0.1};
Point(7) = {2.03, 2.85, 0, 0.1};
Point(8) = {1.97, 2.25, 0, 0.1};
Point(9) = {2.03, 2.25, 0, 0.1};


//+ Slot geometry segments
Line(6) = {7, 6};
Line(7) = {7, 9};
Line(8) = {8, 6};
Line(9) = {9, 8};

//+ Slot geometry contour
Line Loop(3) = {6,-7,-8,-9};

//+ Slot geometry surface
Surface(2) = {3};

//+ Creating slotted disc with boolean difference between circle surface and slot surface geometries
BooleanDifference(3) = { Surface{1}; Delete; }{ Surface{2}; Delete; };

//+ Redefining segment loops
Line Loop(5) = {4, 1, 2, 3};
Line Loop(6) = { 8,-7,6,-5,-9};

//+ Merging surfaces of square and slotted disc
Plane Surface(4) = {5, 6};

//+ Assigning target edge length value of l_{s}=0.1 at points of slotted disc interface
Characteristic Length {5, 8, 7, 6, 9} = 0.1;

//+ Get element sizes by interpolating the values of assigned target edge lengths
Mesh.MeshSizeFromPoints
```

(a)



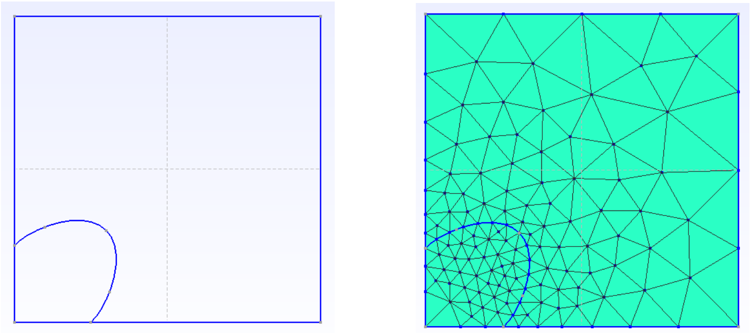
(b) (c)

Figure A.2: Example of generating a graded mesh of the zalesak disc geometry with *Gmsh*©. (a) In the *input.geo* file first the geometry of the square border is created and its points are assigned a target edge length 0.5. Next, the geometry of the circle and the slot is created. The slotted disc is a result of a boolean difference btw the surface of the circle and the slot. The vertices of the slotted disc interface are assigned a target edge length 0.1. The size of the mesh elements will then be computed by interpolating these values of the square's points and the interface's points (Mesh.MeshSizeFromPoints). (b) The geometry of the *input.geo* file. (c) The graded mesh of zalesak disc after calling the command "$input.geo\ -2\ -algo\ del2d\ output.msh$".



```
SetFactory("OpenCASCADE");

//+ lc_boundary is the target edge length assigned to points of the square
//+ lc is the target edge length assigned to points of the parabolic interface
lc_boundary=0.5;
lc=1e-1;

Point(1)={1,-1,0,lc_boundary};
Point(2)={-1,-1,0,lc_boundary};
Point(3)={-1,1,0,lc_boundary};
Point(4)={1,1,0,lc_boundary};

//+ Boundary points that also will be part of the parabolic interface
Point(8)={-1,-0.5,0,lc};
Point(9)={-0.5,-1,0,lc};


//+ Define square segments
Line(1) = {2, 9};
Line(2) = {9, 1};
Line(3)={1,4};
Line(4)={4,3};
Line(5) = {3, 8};
Line(6)={8,2};



//+ Define the parabola with y=a*x^2 with xmax extremes and origin
a = 3;
x_max = 0.3;

Point(5) = {0,0,0,lc};
Point(6) = {x_max , a*x_max*x_max, 0,lc};
Point(7) = {-x_max , a*x_max*x_max, 0,lc};

//+ Translate the origin by x_origin_translation and y_origin_translation and rotate 3*pi/4 around the origin point (z-axis)

x_origin_translation=-0.4;
y_origin_translation=-0.4;

Translate {x_origin_translation, y_origin_translation, 0} {
  Point{5};
  Point{6};
  Point{7};
}

Rotate{{0,0,1},{x_origin_translation,  y_origin_translation, 0},3*Pi/4}{Point{6};Point{7};}



//+ Create parabolic interface
Spline(7) = {8, 6, 5, 7, 9};

//+ Define square contour
Line Loop(1) = {1,2,3,4,5,6};

//+ Define square surface domain
Plane Surface(1) = {1};


//+ Include parabolic interface in square domain
Line{7} In Surface{1};


//+ Get element sizes by interpolating the values of assigned target edge lengths
Mesh.MeshSizeFromPoints
```

(a)



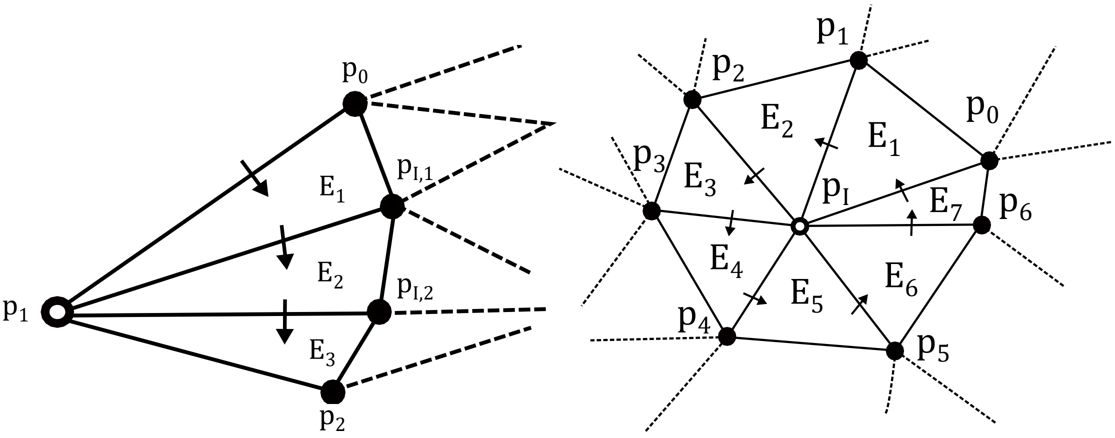
(b) (c)

Figure A.3: Example of generating a graded mesh of the square with the parabolic interface geometry with *Gmsh*©. (a) In the *input.geo* file first the geometry of the square border are created and its assigned a target edge length 0.5. Next, the geometry parabolic interface is created. The slotted disc is a result of a boolean difference btw the surface of the circle and the slot. The vertices of the parabolic interface are assigned a target edge length 0.1. The size of the mesh elements will then be computed by interpolating these values of the square's points and the interface's points (Mesh.MeshSizeFromPoints). (b) The geometry of the *input.geo* file. (c) The graded mesh of the square including the parabolic interface after calling the command "$input.geo\ -2\ -algo\ del2d\ output.msh$".

A.3 Triangulation algorithm

A.3.1 Locking mechanism

During the triangulation algorithm, the vertices V mesh are verified to be included in the set of locked vertices V_{locked}. For a contour vertex, the elements are traversed using the edges that are connected to the vertex. Starting from one of the contour edges linked to the vertex, if the traversal ends with an edge that is also a contour edge, then the vertex is considered locked (Fig. A.4a). For an interior vertex, starting from an edge connected to the interior point the elements are traversed in a similar manner. If the final visited edge of the traversal is the initial one, then the inner vertex is considered locked (Fig. A.4b). A contour facet F is tagged as F_{locked} an no longer available to form connections if the element that contains the facet F contains another contour facet.



(a) (b)

Figure A.4: (a) Example of a locked contour vertex p_1. Starting from the contour facet $\{p_0, p_1\}$ of element E_1 the adjacent by edges of the elements E_2 and E_3 are visited via the traversal of facets $\{p_0, p_1\} \to \{p_1, p_{I,1}\} \to \{p_1, p_{I,2}\} \to \{p_1, p_2\}$. The facet $\{p_1, p_2\}$ is a contour facet. Therefore, the contour vertex p_1 is locked. (b) Exampled of a locked interior vertex. Edges of elements surrounding the interior vertex p_I are traversed. Starting from facet $\{p_I, p_0\}$ this leads to the edge traversal $\{p_I, p_0\} \to \{p_I, p_1\} \to \{p_I, p_2\} \to \{p_I, p_3\} \to \{p_I, p_4\} \to \{p_I, p_5\} \to \{p_I, p_6\} \to \{p_I, p_0\}$. The starting facet $\{p_I, p_0\}$ is also the last visited facet. Therefore, p_I is locked.

A.3.2 Sub-contour detection

The detection of a sub-contour is based on facet traversal from a set F_{visit}. F_{visit} includes a set of facets that are located in the interior of the contour ($F_{interior}$) and contour facets ($F_{contour}$). The algorithm starts with an initial set of vertices that are open for connections (V_{open}) and a set of interior facets. Before performing edge traversal to detect sub-contours the algorithm fist verifies that all the vertices included in V_{open} are not locked due to a creation of an unregistered element (Alg. 11, Lines 9-22). If after the verification, no vertices are included in V_{open} then the algorithm stops (Alg. 11, Lines 23-25). If open vertices still exist, then the interior facets that either are common facet that contain a vertex that is locked or are common facets for two elements are deleted from the list (Alg. 11, Lines 26-35). The updated set of F_{facet} along with the set of open vertices V_{open} are included in a set of visiting vertices V_{visit} and visiting facets F_{visit}. F_{visit} also includes the contour facets. Starting from a vertex in V_{visit} the next vertex v_{next} that is contained in one of the facets F_{visit} is visited. The facet that is traversed is included in the set of a sub-contour facet $F_{subcontour}$, the vertex v_{next} if then removed from the set V_{visit} and the next vertex that is linked to it through a facet from F_{visit} is visited. When the visited vertex is identified as the starting vertex, all the facets that are part of a sub-contour that include the stating vertex have been visited and the sub-contour is included in a sub-contour list (Alg. 11, Lines 38-49). The procedure of acquiring the formed sub-contours is iterated until the set V_{visit} is empty (Fig. A.5).

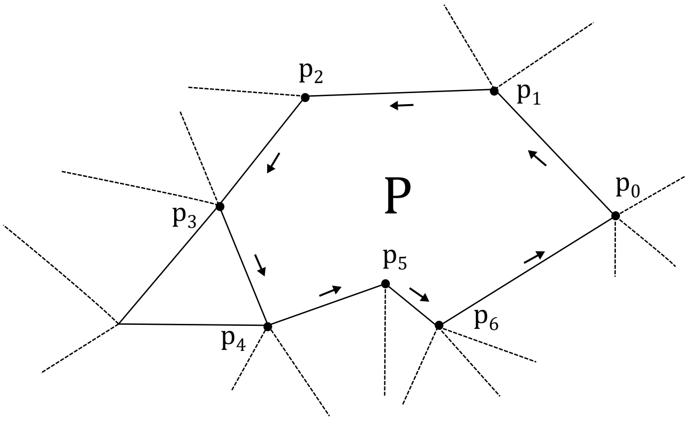


Figure A.5: Example of spotting a sub-contour P. After the connection of of contour facets with element. The vertices (contour or inner) $\{p_0, p_1, p_2, p_3, p_4, p_5, p_6\}$ are open for further connections and are contained in the set of vertices to visit V_{visit}. The list of facets to visit F_{visit} are the facets that link the open vertices. F_{visit} contains facets that are located in the interior of a contour (appearing after the creation of elements) or contour facets. Starting from p_0, $v_{next} = p_1$ is visited through the facet of $F_{visit} = \{p_0, p_1\}$. p_1 is removed from the list V_{visit}. The rest of the vertices from V_{visit} are visited using the facets of F_{visit} in a similar fashion. Each time a vertex is visited it is removed from the set V_{visit}. The traversal stops at p_0 which is the initial visiting vertex. All the edges visited from F_{visit} are contained the the set $F_{contour}$ which contains the facets of sub-contour P.

Algorithm 11: Algorithm for detecting the formation of sub-contours

```
$F_{contour}$: Set of contour facets
$F_{interior}$: Set of facets on the interior of the contour
$V_{locked}$: Set of locked vertices
$V_{open}$: Set vertices open for connection
$F_{subcontour}$: Set of sub-contour facets
$SubContourList$: Sub-contour List
$V_{visit}$: Set of vertices for edge traversal
$F_{visit}$: Set of facets included in traversal
/* Verifying if vertices of $V_{open}$ are locked due to unregistered element */
foreach vertex $v$ in $V_{open}$ do
    if $v$ is locked after creation of an element $E$ then
        delete $v$ from $V_{open}$
    end
    if $v$ is a contour vertex and not in $V_{locked}$ then
        Perform edge traversal of elements connected with $v$ from starting from both
          contour edges linked with $v$
        Gather last indices of the vertices from lasst edges found, $(v_1, v_2)$ found during
          edge traversal from both directions that linked to $v$
        if edge $F = (v_1, v_2)$ is found in the set $F_{interior}$ then
            Add element $E = (F, v)$ to list of elements
            Remove $F$ from $F_{interior}$
            Remove $v$ from $V_{locked}$
        end
    end
end
if $V_{open}$ is empty then
    Return
end
/* Excluding facets from $F_{interior}$ */
foreach edge $F$ in $F_{interior}$ do
    if $F$ has one vertex that belongs in $V_{locked}$ then
        delete $F$ from $F_{interior}$
    end
end
foreach vertex $v$ in $V_{open}$ do
    if $v$ is connected to a facet $F$ in $F_{interior}$ and $F$ is a common facet for two elements
      then
    end
    Remove $F$ from $F_{interior}$
end
/* Detect sub-contours using sets $V_{visit}$ and $F_{visit}$ */
$V_{visit} = V_{open}$
$F_{visit} = F_{contour} + F_{interior}$
foreach $v$ in $V_{visit}$ do
    while $V_{visit}$ is not empty do
        $v_{visit} = v$
        do
            Find facet a $F = (v_{visit}, v_{next})$ from $F_{visit}$ , where $v_{next}$ in $V_{visit}$
            Include $F$ in $F_{subcontour}$
            $v_{next} = v_{visit}$
            Remove $v_{visit}$ from $V_{visit}$
        while $v_{next} \neq v$;
    end
    Include $F_{subcontour}$ in $SubContourList$
end
return $SubContourList$
```

B Code

B.1 Mesh generation

B.1.1 Feature Transformation

```
import numpy as np

# Function that applies procrustes superposition to a polygon
def apply_procrustes(polygon_points, plot=False):

    # Get reference polygon and adjust any random poygon to it
    ref_polygon=get_reference_polygon(polygon_points.shape[0])


    # Mean of each coordinate
    mu_polygon = polygon_points.mean(0)
    mu_ref_polygon = ref_polygon.mean(0)

    # Centralize data to the mean
    centralised_ref_polygon_points = ref_polygon-mu_ref_polygon
    centralised_polygon_points = polygon_points-mu_polygon

    # Squared sum of X-mean(X)
    ss_ref_polygon_points = (centralised_ref_polygon_points**2.).sum()
    ss_polygon_points = (centralised_polygon_points**2.).sum()


    # Frobenius norm of X
    norm_ss_ref_polygon_points = np.sqrt(ss_ref_polygon_points)
    norm_ss_polygon_points = np.sqrt(ss_polygon_points)


    # Scale to equal (unit) norm
    centralised_ref_polygon_points /=norm_ss_ref_polygon_points
    centralised_polygon_points /=norm_ss_polygon_points


    # Finding best rotation to superimpose on regular triangle
    # Applying SVD to the  matrix
    A = np.dot(centralised_ref_polygon_points.T, centralised_polygon_points)
    U,s,Vt = np.linalg.svd(A,full_matrices=False)
    V=Vt.T
    R = np.dot(V,U.T)


    traceTA = s.sum()
    indices=[i for i in range(polygon_points.shape[0])]



    polygon_transformed =norm_ss_ref_polygon_points*traceTA*np.dot(
        centralised_polygon_points,R)+mu_ref_polygon

    if plot==True:
        plot_coords=np.vstack([polygon_transformed,polygon_transformed[0]])
        (s,t)=zip(*plot_coords)
        plt.plot(s,t)
        for index,i in enumerate(indices):
            plt.annotate(str(i),(s[index],t[index]))

    return polygon_transformed
```

B.1.2 Approximation of inner vertices location

B.1.2.1 Point selection

```
import numpy as np

''' Function selecting minimum score(grid_qualities) grid points. The function returns
    the selected points, the surrounding grid points and their scores to conduct
    interpolation. '''
def select_points(contour,grid_points,grid_qualities,nb_of_points,nb_of_grid_points,
    target_edge_length):

    selected_points=[]
    surrounding_points_indices_list=[]
    surrounding_points_list=[]
    grid_qualities_surrounding_list=[]
    grid_qualities_duplicate=grid_qualities.flatten()

    # Select minimum score grid points
    for i in range(nb_of_points):

        minimum_index=np.argmin(grid_qualities_duplicate)
        surrounding_points_index=np.array([minimum_index+1,minimum_index-1,
            minimum_index+nb_of_grid_points
        ,minimum_index-nb_of_grid_points,minimum_index-nb_of_grid_points+1,minimum_index-nb_of_grid_points-1,
        minimum_index+nb_of_grid_points+1,minimum_index+nb_of_grid_points-1])


        surrounding_points_index_2_ring=np.array([minimum_index+2,minimum_index-2,
        minimum_index+2*nb_of_grid_points,
        minimum_index+2*nb_of_grid_points+2,minimum_index+2*nb_of_grid_points-2,
        minimum_index+2*nb_of_grid_points+1,minimum_index+2*nb_of_grid_points-1,


        minimum_index-2*nb_of_grid_points,
        minimum_index-2*nb_of_grid_points+1,minimum_index-2*nb_of_grid_points-1,
        minimum_index-2*nb_of_grid_points+2, minimum_index-2*nb_of_grid_points-2 ,


        minimum_index+nb_of_grid_points+2,
        minimum_index+nb_of_grid_points-2,

        minimum_index-nb_of_grid_points+2,
        minimum_index-nb_of_grid_points-2



        ])
        surrounding_points_index_3_ring=np.array([
        minimum_index+3,minimum_index-3,

        minimum_index+3*nb_of_grid_points,
        minimum_index+3*nb_of_grid_points+3,minimum_index+3*nb_of_grid_points-3,
        minimum_index+3*nb_of_grid_points+2,minimum_index+3*nb_of_grid_points-2,
        minimum_index+3*nb_of_grid_points+1,minimum_index+3*nb_of_grid_points-1,



        minimum_index-3*nb_of_grid_points,
        minimum_index-3*nb_of_grid_points+3, minimum_index-3*nb_of_grid_points-3 ,
        minimum_index-3*nb_of_grid_points+2, minimum_index-3*nb_of_grid_points-2 ,
        minimum_index-3*nb_of_grid_points+1,minimum_index-3*nb_of_grid_points-1,



        minimum_index+nb_of_grid_points+3,
        minimum_index+nb_of_grid_points-3,

        minimum_index-nb_of_grid_points+3,
        minimum_index-nb_of_grid_points-3,

        minimum_index+2*nb_of_grid_points+3,
        minimum_index+2*nb_of_grid_points-3,

        minimum_index-2*nb_of_grid_points+3,
        minimum_index-2*nb_of_grid_points-3




        ])


        try:
            surrounding_points=grid_points[np.asarray(surrounding_points_index)]
            surrounding_points_2_ring=grid_points[np.asarray(surrounding_points_index_2_ring)]
            surrounding_points_3_ring=grid_points[np.asarray(surrounding_points_index_3_ring)]
        except IndexError as e:
            print(e)

    point_minimum=grid_points[minimum_index]
    selected_points.append(np.array(point_minimum))

    # Surrounding region (ring) in accordance with 20% of target edge length
    if .6<target_edge_length<=1:
        ring=3
    elif .4<target_edge_length<.6:
        ring=2
    else:
        ring=1

    grid_qualities_duplicate[minimum_index]=100
    if ring==3:
        grid_qualities_duplicate[np.asarray(surrounding_points_index)]=100
        grid_qualities_duplicate[np.asarray(surrounding_points_index_2_ring)]=100
        grid_qualities_duplicate[np.asarray(surrounding_points_index_3_ring)]=100
        surrounding_points_index=np.append(surrounding_points_index_2_ring,np.append(surrounding_points_index,minimum_index))
        surrounding_points_index=np.append(surrounding_points_index,surrounding_points_index_3_ring)
        surrounding_points_list.append(surrounding_points_3_ring,(np.append(surrounding_points_2_ring,np.append(surrounding_points,point_minimum)))),grid_qualities_surrounding_list.append(grid_qualities.flatten()[np.asarray(surrounding_points_index)]  )
    if ring==2:
        grid_qualities_duplicate[np.asarray(surrounding_points_index)]=100
        grid_qualities_duplicate[np.asarray(surrounding_points_index_2_ring)]=100
        surrounding_points_index=np.append(surrounding_points_index_2_ring,np.append(surrounding_points_index,minimum_index))
        surrounding_points_list.append(surrounding_points_2_ring)
        surrounding_points_list.append(np.append(surrounding_points,point_minimum))
        surrounding_points_list.append(grid_qualities_surrounding_list.append(grid_qualities.flatten()[np.asarray(surrounding_points_index)]))
    else:
        surrounding_points_index=np.append(surrounding_points_index,minimum_index)
        surrounding_points_indices_list.append(surrounding_points_index)
        surrounding_points=np.append(surrounding_points,point_minimum)
        surrounding_points_list.append(surrounding_points), grid_qualities_surrounding_list.append(grid_qualities.flatten()[np.asarray(surrounding_points_index)])
        grid_qualities_duplicate[np.asarray(surrounding_points_index)]=100


    return np.array(selected_points),np.array(surrounding_points_list),np.array(grid_qualities_surrounding_list)
```

B.1.2.2 Interpolation

```
from scipy import interpolate


# Override interp2d to include quadratic spline
class quadratic_bspline(interpolate.interp2d):
    def __init__(self,*args,**kws):
        try:
            super(quadratic_bspline, self).__init__(*args,**kws)

        except ValueError:
            kx=ky=2
            x=args[0]
            y=args[1]
            z=args[2]
            rectangular_grid = (z.size == len(x) * len(y))
            if rectangular_grid:
                self.tck = scipy.interpolate.fitpack.bisplrep(x, y, z, kx=kx, ky=ky, s=0.0)
            else:
                nx, tx, ny, ty, c, fp, ier = scipy.interpolate.dfitpack.regrid_smth(
                x, y, z, None, None, None, None,
                kx=kx, ky=ky, s=0.0)
                self.tck = (tx[:nx], ty[:ny], c[:(nx - kx - 1) * (ny - ky - 1)],
                kx, ky)
        self.bounds_error = False
        self.fill_value = None
        self.x, self.y, self.z = [np.array(a, copy=copy) for a in (x, y, z)]

        self.x_min, self.x_max = np.amin(x), np.amax(x)
        self.y_min, self.y_max = np.amin(y), np.amax(y)



''' Function that applies interpolation to a neighbor region around the selected grid points '''
def bilineaire_interpolation(surrounding_points,grid_qualities_surrounding,selected_point):
    size=int(int(len(surrounding_points))/2)
    surrounding_points=surrounding_points.reshape(size,2)


    z= grid_qualities_surrounding.reshape(int(sqrt(size)),int(sqrt(size)))

    # Define quadratic b-spline of surrounding region
    z_interp = interpolate.interp2d(surrounding_points[:,0].reshape(int(sqrt(size)),int(sqrt(size))),surrounding_points[:,1].reshape(int(sqrt(size)),int(sqrt(size))),z,kind='quintic')

    x_new=np.linspace(min(surrounding_points[:,0]),max(surrounding_points[:,0]),100)
    y_new=np.linspace(min(surrounding_points[:,1]),max(surrounding_points[:,1]),100)
    z_new=z_interp(x_new,y_new)
    epsilon=1e-4
    bnds=((min(surrounding_points[:,0]),max(surrounding_points[:,0])),(min(surrounding_points[:,1]),max(surrounding_points[:,1])))

    # Find minimum that approximates inner vertex location
    minimum=minimize(lambda v: z_interp(v[0],v[1]), np.array([selected_point[0]+epsilon,selected_point[1]+epsilon]), method='TNC',bounds=bnds)
    return np.array([minimum.x[0],minimum.x[1]])
```

B.1.3 Triangulation algorithm

B.1.3.1 Vertex locking mechanism

```
# Finds element containing the edge and exits (does not give the full list of elements)
def edge2elem(edge,set_of_elements):
    Found_element=()
    Remaining_edge=()

    for element in set_of_elements.copy():
        if edge[0] in  set(element) and edge[1] in element:
            Found_element=element
            Remaining_index=set(element)-set(edge)
            Remaining_index=list(Remaining_index)
            Remaining_edge=(edge[0],Remaining_index[0])
            break
        else:
            Found_element=None
            Remaining_edge=None
    return  Remaining_edge,Found_element




# Checking if a contour vertex is closed (locked)
def is_closed_ring(vtx,set_of_elements,*adj_vtx):
    contour_edge1=(vtx,adj_vtx[0])
    contour_edge2=(vtx,adj_vtx[1])
    visited_elements=set_of_elements.copy()

    target_edge=contour_edge1

    edges_found=[]
    edges_found.append(contour_edge1)

    proceed=True

    while proceed:

        if not visited_elements:
            break

        remaining_edge,found_element=edge2elem(target_edge,visited_elements)

        if found_element is None:
            proceed=False
            break

        visited_elements.remove(found_element)
        edges_found.append(remaining_edge)
        target_edge=remaining_edge




    found_contour_edge1,found_contour_edge2=False,False
    found_contour_edges=False

    # Checking if both contour edges  contained in the set of edges acquired
    for edge in edges_found:
        condition1= contour_edge1[0] in set(edge) and contour_edge1[1] in set((edge))
        condition2= contour_edge2[0] in set(edge) and contour_edge2[1] in set((edge))
        if condition1:
            found_contour_edge1=True
        if condition2:
            found_contour_edge2=True

    if found_contour_edge1 and found_contour_edge2:
        found_contour_edges=True


    visited_elements.clear()
    return edges_found,found_contour_edges
```

```




# Checking if a vertex inside the contour is closed (locked)
def is_closed_interior_point(interior_point,set_of_interior_edges,set_of_elements):
    is_closed = False
    print("Checking if interior point {} is closed".format(interior_point))
    found_edge=False
    for edge in set_of_interior_edges:
        if interior_point in edge:
            for index in edge:
                if index!=interior_point:
                    first_found_index= index
                    found_edge=True
                print("found {} in {}".format(interior_point,edge))
                break

    # the interior is not linked with any point
    if not found_edge:
        return is_closed

    keep_looking=True
    visited_elements=set()
    while keep_looking:
        for index,element in enumerate(set_of_elements):
            if set(edge).issubset(set(element)) and element not in visited_elements:
                visited_elements.add(element)
                found_index=[int(i) for i in set(element)-set(edge)]
                print("found index {} in element {} ".format(found_index,element))

                # Change edge value
                lst=list(edge)
                lst=[interior_point,found_index[0]]
                edge=tuple(lst)
                if found_index==first_found_index:
                    is_closed=True
                    keep_looking=False
                    break
            elif not set(edge).issubset(set(element)) and index==len(set_of_elements)-1:
                break
            keep_looking=False
            print("Interior vertex {} is open".format(interior_point))



    return is_closed
```

B.1.3.2 Spotting sub-contours

```
import numpy as np
import torch
from  more_itertools import unique_everseen
from collections import OrderedDict

```

```
# Function to check if element with edge1 and edge2 exists
def found_element_with_edges(edge1,edge2,set_elements):
    found_element=False
    edge1=set(edge1)
    edge2=set(edge2)
    possible_element=edge1.union(edge2)
    for element in set_elements:
        if set(element)==set(possible_element):
        found_element=True
    return found_element



# Function to check if the vertices of a contour edges are linked with an inner point
def linked_via_inner_point(vtx1,vtx2,edges_to_visit,set_of_open_vertices):

    vtx_set=set([vtx for edges in edges_to_visit  for vtx in edges])

    if vtx1 not in vtx_set and vtx2 not in vtx_set:
        return True

    for edges in edges_to_visit:
        for index,vtx in enumerate(edges):
            if edges[index]==vtx1:
                adjacent_point=edges[(index+1)%2]
                    for edges in edges_to_visit:
                        for index,vertices in enumerate(edges):
                            if edges[index]==adjacent_point and edges[(index+1)%2]==
                              vtx2:
                                return True
    return False



# Find elements that are connected to a specific vertex
def vert2elem(vtx,set_of_elements):
    found_elements=set()
    for element in set_of_elements:
        if vtx in  set(element):
            found_elements.add(element)
    return found_elements

# Find edges that are connnected to a specifix vertex
def edge2vert(vtx,polygon,set_interior_edges):
found_edges=set()
    if vtx<polygon.shape[0]:
        found_edges.add((vtx,(vtx+1)%polygon.shape[0]))
        found_edges.add((vtx,(vtx-1)%polygon.shape[0]))
    for edge in set_interior_edges:
        if vtx in set(edge):
            if edge not in found_edges or edge[::-1] not in found_edges:
                found_edges.add(edge)
    return found_edges



# Sort edges around point counterclock wise #
def sort_edges_around_vertex(vertex,edges_around_vert,polygon,points):
    polygon_with_points=np.vstack([polygon,points])
    edges_coordinates=[]
    edges_indices=[]
    for edges in edges_around_vert:
        edge_indices=np.asarray(edges if edges[0]==vertex else edges[::-1])
        edges_indices.append(edges if edges[0]==vertex else edges[::-1])
        edges_coordinates.append(polygon_with_points[edge_indices])

    edge_list={edge:edge_coordinate for edge,edge_coordinate in zip(edges_indices,edges_coordinates)}
    vertex_list={edge:edge_coordinates[1]-edge_coordinates[0] for edge,edge_coordinates in zip(edge_list.keys(),edges_coordinates)}
    vertex_coordinates=list(vertex_list.values())
    angle_list=[]
    for vertices in vertex_coordinates:
        angle=angle_counterclockwise(np.array([0,1]),vertices)
        angle_list.append(angle)


    angle_list={edge:angle for edge,angle in zip(edges_indices,angle_list)}
    sorted_edges=dict(OrderedDict(sorted(angle_list.items(),key=lambda x:x[1])))
    print(edge_list)
    print(sorted_edges)
    return [*sorted_edges]


''' Function that performs edge traversal starting from a vertex (starting_vertex) based on the current set of open vertices, the set of elements and initial set of edges that can be visited. Additional edges to visit (candidate_edge) are added in the process to find spot sub contour. Adjacent edges (initial_pair_of_adjacent_edges, pair_of_adjacent_edges) represent edges that belong in the same sub contour and are meant to be visited one after another if a vertex is part of multiple sub contours (set_of_common_vertices) represents a set containing such vertices).'''

def polygon_2_vtx(starting_vertex,set_of_elements,initial_edges_to_visit,edges_to_visit,set_of_common_vertices,initial_pair_of_adjacent_edges,pair_of_adjacent_edges,set_of_open_vertices,set_orphan_vertices,polygon):


    if not edges_to_visit:
        return
    added_edges=set()
    print(" Initial edges to visit" , edges_to_visit)

    # Check for candidate edges that could be included in the set of edges to visit
    if len(set_of_common_vertices)==0:
     # if vertex is a contour vertex
     if vtx<polygon.shape[0]:
        candidate_point1=(vtx+1)%polygon.shape[0]
        if not linked_via_inner_point(vtx,candidate_point1,edges_to_visit,set_of_open_vertices):
            is_ok=True
            candidate_edge1=(vtx,candidate_point1)
        for element in set_of_elements:
         if set(candidate_edge1).issubset( set(element)):
               is_ok=False
         if candidate_edge1 in edges_to_visit.copy() or candidate_edge1[::-1] in edges_to_visit.copy():
            is_ok=False




        for edge in list(edges_to_visit):
            if candidate_edge1[0] in edge or candidate_edge1[1] in edge:
                Found=True
                break


        if not Found:
            is_ok=False

        if is_ok and candidate_edge1[0] in set_of_open_vertices and candidate_edge1[1] in set_of_open_vertices:
            edges_to_visit.add(candidate_edge1)
            added_edges.add(candidate_edge1)
            candidate_point2=(vtx-1)%polygon.shape[0]
        if not linked_via_inner_point(vtx,candidate_point2,edges_to_visit,set_of_open_vertices):
            is_ok=True


        candidate_edge2=(vtx,candidate_point2)
        for element in set_of_elements:
            if set(candidate_edge2).issubset( set(element)):
                is_ok=False

        if candidate_edge2 in edges_to_visit.copy() or candidate_edge2[::-1] in edges_to_visit.copy():
            is_ok=False

        Found=False
        for edge in list(edges_to_visit):
            if candidate_edge2[0] in edge or candidate_edge2[1] in edge:
                Found=True
                break

        if not Found:
            is_ok=False

        if is_ok and candidate_edge2[0] in set_of_open_vertices and candidate_edge2[1] in set_of_open_vertices:
            edges_to_visit.add(candidate_edge2)
            added_edges.add(candidate_edge2)

    else:
        # if set of commong vertices is not empty

        vertex_list=set([vtx for edges in pair_of_adjacent_edges for edge in edges for vtx in edge])-set_of_common_vertices

        for vtx in vertex_list:
            candidate_point1=(vtx+1)%polygon.shape[0]
            is_ok=True
            if not linked_via_inner_point(vtx,candidate_point1,initial_edges_to_visit,set_of_open_vertices):
                if (vtx,candidate_point1) in edges_to_visit.copy() or(vtx,candidate_point1) in  edges_to_visit.copy():
                    for edges_in_same_polygon in initial_pair_of_adjacent_edges:
                        if (vtx,candidate_point1) in edges_in_same_polygon or (candidate_point1,vtx) in edges_in_same_polygon:
                            is_ok=False
            if is_ok:
                edges_to_visit.add((vtx,candidate_point1))
                added_edges.add((vtx,candidate_point1))


            candidate_point2=(vtx-1)%polygon.shape[0]
            if not linked_via_inner_point(vtx,candidate_point2,initial_edges_to_visit,set_of_open_vertices):
                if (vtx,candidate_point2) in edges_to_visit.copy() or(vtx,candidate_point2) in  edges_to_visit.copy():
                    for edges_in_same_polygon in initial_pair_of_adjacent_edges:
                        if (vtx,candidate_point2) in edges_in_same_polygon or (candidate_point2,vtx) in edges_in_same_polygon:
                            is_ok=False
            if is_ok:
                edges_to_visit.add((vtx,candidate_point2))
                added_edges.add((vtx,candidate_point2))

        for vtx in set_of_open_vertices:
            if vtx<polygon.shape[0]:
                candidate_point1=(vtx+1)%polygon.shape[0]
```

```
if not linked_via_inner_point(vtx,candidate_point1,edges_to_visit,set_of_open_vertices):
    is_ok=True
candidate_edge1=(vtx,candidate_point1)
for element in set_of_elements:
    if set(candidate_edge1).issubset( set(element)):
        is_ok=False
if candidate_edge1 in edges_to_visit.copy() or candidate_edge1[::-1] in edges_to_visit.copy():
    is_ok=False
for edges_in_same_polygon in initial_pair_of_adjacent_edges:
    if candidate_edge1 in edges_in_same_polygon or candidate_edge1[::-1] in edges_in_same_polygon:
        is_ok=False

Found=False
for edge in list(edges_to_visit):
    if candidate_edge1[0] in edge or candidate_edge1[1] in edge:
        Found=True
        break


if not Found:
    is_ok=False

if is_ok and candidate_edge1[0] in set_of_open_vertices and candidate_edge1[1] in set_of_open_vertices:
    edges_to_visit.add(candidate_edge1)
    added_edges.add(candidate_edge1)



candidate_point2=(vtx-1)%polygon.shape[0]
if not linked_via_inner_point(vtx,candidate_point2,edges_to_visit,set_of_open_vertices):
    is_ok=True
candidate_edge2=(vtx,candidate_point2)
for element in set_of_elements:
    if set(candidate_edge2).issubset( set(element)):
        is_ok=False
if candidate_edge2 in edges_to_visit.copy() or candidate_edge2[::-1] in edges_to_visit.copy():
    is_ok=False
for edges_in_same_polygon in initial_pair_of_adjacent_edges:
    if candidate_edge2 in edges_in_same_polygon or candidate_edge2[::-1] in edges_in_same_polygon:
        is_ok=False

Found=False
for edge in list(edges_to_visit):
    if candidate_edge2[0] in edge or candidate_edge2[1] in edge:
        Found=True
```

```
                        break


                if not Found:
                    is_ok=False

                if is_ok and candidate_edge2[0] in set_of_open_vertices and candidate_edge2[1] in set_of_open_vertices:
                    edges_to_visit.add(candidate_edge2)
                    added_edges.add(candidate_edge2)

    # After adding candidate edges to the set of edges to visit commence traversal
     print("Edges to visit:",edges_to_visit)
    subpolygon=[]

    set_of_points=set([j for i in edges_to_visit for j in i])

    if starting_vertex not in set_of_points:
        return

    found_vertex=starting_vertex
    target_edge=[]
    visited_added_edge=False
    deleted_edges=set()
    deleted_adjacent_edges=set()

    count=0
    while not closed:
        if len(edges_to_visit)==0:
            return 0

        count+=1
        for index,edge in enumerate(edges_to_visit.copy()):
            visiting_vertex=found_vertex

            ''' target edge represents an edge that needs to be followed in the set to remain in the same sub-contout '''
            if target_edge:
                if edge!= target_edge[0] and  edge!= tuple(reversed(target_edge[0])) :
                    continue
            else:
                target_edge.pop()

             if visiting_vertex not in set(edge):
                if len(edges_to_visit)==0 and visited_added_edge or index==int(len(edges_to_visit))-1 and visited_added_edge:
                    print(" Reached end found no matching vertex after visiting added edge ")
                for edge in deleted_edges:
                    edges_to_visit.add(edge)
                for edge in deleted_adjacent_edges:
                    pair_of_adjacent_edges.add(edge)
        return 0

        subpolygon.append(visiting_vertex)

        print(visiting_vertex," in ", edge)

        ''' Starting from a visiting vertex may not be a good idea because we do not know if it will be included to close a polygon'''
        if (edge in added_edges or edge[::-1] in added_edges) and count==1:
                continue


        for index in set(edge):
            if visiting_vertex!= index:
                found_vertex=index
                print("Found vertex:",found_vertex)
                subpolygon.append(found_vertex)

        found_crossroad=False
        found_in_set=False

    ''' Check if edge is part of a crossroad (check if found vertex is point of multiple polygons). If it is, then the next visiting edge should be the one is the pair of adjacent edges  '''
        if found_vertex in set_of_common_vertices:
            found_crossroad=True

        ''' A duplicate edge is an edge that is common for multiple subcontours, and should not be deleted from the list of edges to visit after beeing visited once'''
        duplicate_edge=False

        if found_crossroad:

            for  edges_in_same_polygon in pair_of_adjacent_edges.copy():
                if edge in set(edges_in_same_polygon) or tuple(reversed(edge)) in set(edges_in_same_polygon):
                    for edges in edges_in_same_polygon:
                        if found_vertex in edges_in_same_polygon[0] and found_vertex in edges_in_same_polygon[1]:
                        if edges!=edge and edges!=tuple(reversed(edge)) and edges not in deleted_edges and tuple(reversed(edges)) not in deleted_edges:
                            target_edge.append(edges)
                            found_in_set=True
                            print("edge {} should be followed by {}".format(edge, edges))
                            count=0

                            deleted_adjacent_edges.add(edges_in_same_polygon)
                            pair_of_adjacent_edges.discard(edges_in_same_polygon)

```

```
                                for edges_in_same_polygon in pair_of_adjacent_edges:
                                    for edges in edges_in_same_polygon:
                                        if edge==edges or edge[::-1]==edges:
                                            count+=1
                                        if count>1:
                                            print("found duplicate edge ",edge)
                                            duplicate_edge=True
                                        break
                    if found_in_set:
                        break




            if not duplicate_edge :
                print("Removing edge",edge)
                if edge in added_edges:
                    visited_added_edge=True
                if edge not in added_edges:
                    deleted_edges.add(edge)
                    edges_to_visit.discard(edge)

            print(edges_to_visit)
            if found_vertex==starting_vertex:
                subpolygon=list(unique_everseen(subpolygon))
                print("Back to starting vertex")
                closed=True
                break

    if len(subpolygon)<3:
        return
    else:
        return subpolygon

''' Fucntion that returns a list of sub-contours (sub_polygon) of the current mesh (contour could be partially meshed) based on the set inner vertices that are not connected (set_orphan_vertices), the set of open vertices (set_of_open_vertices), the set of interior edges (set_of_interior_edges), and the set of elements that are currently formed (set_of_elements). The function also initially includes a list of edges that will be included in a traversal to spot sub-contours.'''
def check_for_sub_polygon(set_orphan_vertices,set_of_open_vertices,set_of_interior_edges,set_of_elements,polygon,points):


    set_polygon_edges=set(tuple(i) for i in get_contour_edges(polygon))

    if not set_of_open_vertices or len(set_of_open_vertices)<3:
        return []

    sub_polygon_list=[]
    modified_interior_edge_set=set_of_interior_edges.copy()

```

```
polygon_connectivity=[tuple(i) for i in get_contour_edges(polygon)]


# Taking care of vertices that are locked but the element is not seen

set_of_unfound_locked_vertices=set()
continue_looking=True

while continue_looking:

    if not set_of_open_vertices :
        continue_looking=False

    set_of_open_vertices_copy=set_of_open_vertices

 for vtx in set_of_open_vertices_copy :

# Check if interior vertex is locked
    found_locked_vtx=False
    if vtx>=polygon.shape[0]:
        is_closed=is_closed_interior_point(vtx,set_of_interior_edges,set_of_elements)
        if is_closed:
            set_of_open_vertices.discard(vtx)
            print("vtx {} is closed after all".format(vtx))
            continue_looking=False
        else:
            continue_looking=False
        break
     # find indices connected to vertices

    vtx1,vtx2 =connection_indices(vtx,get_contour_edges(polygon))


    # Traverse from starting from both edges connected to the vertex
    found_edges1,isclosed1=is_closed_ring(vtx,set_of_elements,vtx2,vtx1)
    found_edges2,isclosed2=is_closed_ring(vtx,set_of_elements,vtx1,vtx2)
    print("Examining if vtx {} is locked".format(vtx))

    # If the contour vertex is locked remove it from set of open vertices
    if isclosed1 or isclosed2:
        print(vtx,"locked after all")
        continue_looking=True

        set_of_open_vertices.discard(vtx)
        for edge in modified_interior_edge_set.copy():
            if vtx in edge:
                modified_interior_edge_set.discard(edge)
        break
    # Checking if there is an element formed but not yet discovered

    # Gather in edges that could connect edges from the edge ring from both sides
```

```
        for edge in found_edges1:
            if edge in polygon_connectivity or edge[::-1] in polygon_connectivity:
                found_edges1.remove(edge)
        for edge in found_edges2:
            if edge in polygon_connectivity or edge[::-1] in polygon_connectivity:
                found_edges2.remove(edge)
        between_edges=[]
        for edge in found_edges1:
            for indices in edge:
                if indices==vtx:
                    continue
            between_edges.append(indices)
        for edge in found_edges2:
            for indices in edge:
                if indices==vtx:
                    continue
            between_edges.append(indices)

        for edge in set_of_interior_edges.copy():
            found_locked_vtx=False

            ''' If an edge is btw the edge ring of the vertex then the vertex is locked and a new elemet is added'''
            if set(between_edges)==set(edge):
                print(vtx,"locked after all")
                found_locked_vtx=True
                set_of_unfound_locked_vertices.add(vtx)


                    if edge in set_of_interior_edges or edge[::-1] in set_of_interior_edges:
                    #modified_interior_edge_set.discard(edge)
                    #print(edge,"removed")
                    #modified_interior_edge_set.discard(edge[::-1])
                    modified_interior_edge_set.discard((vtx,between_edges[0]))
                    modified_interior_edge_set.discard((between_edges[0],vtx))


                    modified_interior_edge_set.discard((vtx,between_edges[1]))
                    modified_interior_edge_set.discard((between_edges[1],vtx))
                    element=(vtx,between_edges[0],between_edges[1])
                    print("Removed:",(vtx),"from set of open vertices")

                    print("Added new element:",element)
                    print("Removed:",(vtx,between_edges[0]),"from set of edges")
                    print("Removed:",(vtx,between_edges[1]),"from set of edges")

                    set_of_elements.add(element)
                    print("New set of elements",set_of_elements)
                    set_of_open_vertices.discard(vtx)

            if found_locked_vtx:
                    continue_looking=True
                    print("Re-evaluting set of open vertices")
                    break
                else:
                    continue_looking=False

             if found_locked_vtx:
                break

    print("set of open vertices",set_of_open_vertices)

    # If the set of open vertices is empty then there is no subpolygon.
    if not set_of_open_vertices or  len(set_of_open_vertices)<3:
        return []

    # In the set of open vertices there may be vertices that are part of multiple sub-contours.
    # Checking if edge that is connected to vertex belongs to two element first.
    # Discard from set of interior edges edges that are common for two elements

    set_of_common_vertices=set()
    pair_of_adjacent_edges=set()
    for vertex in set_of_open_vertices:
        nb_of_polygon=0
        count=0
        for edge in modified_interior_edge_set.copy():
            counter2=0
            if vertex in set(edge):
                count+=1
            for element in set_of_elements:
                if set(edge).issubset(set(element)):
                    counter2+=1
            if counter2==2:
                print("Edge {} is common for two elements".format(edge))
                count-=1
                modified_interior_edge_set.discard(edge)

    # if count is bigger than 2 it means that the vertex is crossroad for multiple subcontours
    if count>=2:
        # Get list of adjacent vertices to the open vertex
        adj_vertices=sorted(list(vtx for edge in modified_interior_edge_set if vertex in set(edge) for  vtx in edge if vtx!=vertex))

         # Checking if vertices are linked , if they are then aren't part of the same polygon
        for index,_ in enumerate(adj_vertices.copy()):


            edge=tuple((adj_vertices[index],adj_vertices[(index+1)%len(adj_vertices)]))

             # Connections could form elements that are not discovered
            if ((edge in set_of_interior_edges or tuple(reversed(edge)) in set_of_interior_edges )and
            ((vertex,edge[0]) in set_of_interior_edges or tuple(reversed((vertex,edge[0]))) in set_of_interior_edges) and
            ((vertex,edge[1]) in set_of_interior_edges or tuple(reversed((vertex,edge[1]))) in set_of_interior_edges) ):
                print("Found new element:",(vertex,edge[0],edge[1]))
                print("({},{}) and ({},{}) are part of the same element".format(edge[0],vertex,edge[1],vertex))
                pair_of_adjacent_edges.add((((edge[0],vertex),(edge[1],vertex))))
            continue

            elements_around_vertex=vert2elem(vertex,set_of_elements)
            edges_around_vertex=edge2vert(vertex,polygon,set_of_interior_edges)
            edge_star=sort_edges_around_vertex(vertex,edges_around_vertex,polygon,points)
            print("edges star of common vertex" ,vertex, "is : ",edge_star)
            print("elements around {} are {}".format(vertex,elements_around_vertex))



            for edge1 in edge_star:
                position_of_edge1=edge_star.index(edge1)

                for edge2 in edge_star:
                    if edge2==edge1:
                    continue


                position_of_edge2=edge_star.index(edge2)
                if (abs(position_of_edge1-position_of_edge2)==1  or abs(position_of_edge1-position_of_edge2)==len(edge_star)-1) and not found_element_with_edges(edge1,edge2,elements_around_vertex):
                    if (edge1,edge2) not in pair_of_adjacent_edges and (edge2,edge1) not in pair_of_adjacent_edges and (tuple(reversed(edge1)),tuple(reversed(edge2))) not in pair_of_adjacent_edges and (tuple(reversed(edge2)),tuple(reversed(edge1))) not in pair_of_adjacent_edges:
                        if (edge1  not in set_polygon_edges and edge1[::-1] not in set_polygon_edges) or (edge2 not in set_polygon_edges and edge2[::-1]  not in set_polygon_edges):
                            print(edge1,"is in  the same polygon with", edge2)

                            pair_of_adjacent_edges.add(((edge2,edge1)))
                            if edge2 not in modified_interior_edge_set and edge2[::-1] not in modified_interior_edge_set:
                                modified_interior_edge_set.add(edge2[::-1])
                            if edge1 not in modified_interior_edge_set and edge1[::-1] not in modified_interior_edge_set:
                                modified_interior_edge_set.add(edge1[::-1])

        '''The set of common vertices includes vertices that are part of more than one sub-contour'''
```

```
         set_of_common_vertices.add(vertex)

    # if the set found is less than 4 then now polygon is formed
    if len(set_of_open_vertices)<4:
        return []

    ''' After cleaning up edges from the interior set include them in the set of of edges to visit to spot sub-contours'''
    edges_to_visit=modified_interior_edge_set



    sub_polygon_list=[]
    initial_edges_to_visit=copy.deepcopy(edges_to_visit)
    initial_pair_of_adjacent_edges=copy.deepcopy(pair_of_adjacent_edges)

    try:
        if set_of_common_vertices:
            for vtx in set_of_common_vertices:
                subpolygon=polygon_2_vtx(starting_vertex,set_of_elements,initial_edges_to_visit,edges_to_visit,set_of_common_vertices,initial_pair_of_adjacent_edges,pair_of_adjacent_edges,set_of_open_vertices,set_orphan_vertices,polygon)
            if subpolygon is not None and subpolygon is not 0:
                sub_polygon_list.append(subpolygon)
            if subpolygon is 0:
                continue

    print(sub_polygon_list)
    except:
    print("Failed")

    while edges_to_visit:
        for vtx in set_of_open_vertices.copy():
            print("Starting with vertex",vtx)
            subpolygon=polygon_2_vtx(vtx,set_of_elements,initial_edges_to_visit,edges_to_visit,set_of_common_vertices,initial_pair_of_adjacent_edges,pair_of_adjacent_edges,set_of_open_vertices,set_orphan_vertices,polygon)
            if subpolygon is not None and subpolygon is not 0:
                sub_polygon_list.append(subpolygon)
            if subpolygon is 0:
                continue

    return sub_polygon_list
```

B.1.3.3 Triangulation

```
import torch

# Function checking if a contour indices are counter clock wise
def is_counterclockwise(polygon):
    area = 0
    counterclokwise=False
    for index,_ in enumerate(polygon):
        second_index=(index+1)%len(polygon)
        area+=polygon[index][0]*polygon[second_index][1]
        area-=polygon[second_index][0]*polygon[index][1]
    if area/2<0:
        counterclokwise=False
    else:
        counterclokwise=True

    return counterclokwise



# Function checking if a point is inside a contour
def ray_tracing(x,y,poly):
    n = len(poly)
    inside = False
    p2x = 0.0
    p2y = 0.0
    xints = 0.0
    p1x,p1y = poly[0]
    for i in range(n+1):
        p2x,p2y = poly[i % n]
            if y > min(p1y,p2y):
                if y <= max(p1y,p2y):
                    if x <= max(p1x,p2x):
                        if p1y != p2y:
                            xints = (y-p1y)*(p2x-p1x)/(p2y-p1y)+p1x
                        if p1x == p2x or x <= xints:
                            inside = not inside
            p1x,p1y = p2x,p2y
    return inside







# Function that calls connectivity network to predict entries of the connection table(quality_matrix)
@torch.no_grad()
def get_quality_matrix_NN(polygon,inner_points):

    nb_of_edges=len(polygon)
    nb_of_inner_points=len(inner_points)

    # Load trained connectivity network
     with open('../network_datasets/connectivity_NN/'+str(nb_of_edges)+'_'+str(nb_of_inner_points)+'_NN_qualities.pkl','rb') as f:
        connection_network=pickle.load(f)
```

```python

    procrustes = apply_procrustes(polygon)
    procrustes_inner_points = apply_procrustes(inner_points)

    input = torch.tensor(np.concatenate([np.asarray(np.append(procrustes, procrustes[0][None,:], axis=0), dtype=np.float32), np.asarray(procrustes_inner_points, dtype=np.float32)], axis=0)[None,None,:,:])

    quality_matrix= net(input)

    quality_matrix = quality_matrix[0].numpy().reshape([procrustes.shape[0], procrustes.shape[0]+len(inner_points)])

    return quality_matrix


''' Triangulation function that returns list of elements based on an ordered connection table (ordered_quality_matrix) '''
def triangulate(polygon,points,ordered_quality_matrix,recursive=True,plot_mesh=True):

    set_edges=set(tuple(i) for i in get_contour_edges(polygon))
    contour_edges=set(tuple(i) for i in get_contour_edges(polygon))
    interior_edges=set()
    set_elements=set()
    set_locked_vertices=set()
    set_orphan_vertices=set()
    set_interior_edge_with_inner_point=set()
    print("initial set edges:", set_edges)


    polygon_with_points=np.vstack([polygon,points])


    print("meshing polygon:", polygon," with inner points :", points)


    ''' Go through the edges of the ordered connection table dictiionary(represent by the keys of the dictionary. The copy of the table is iterated since it can change size due to deletion of entries from spotting locked facets '''
    for edge in ordered_quality_matrix.copy().keys():

        ''' If error is raised the edge entry was deleted for beeing locked. We proceed to next edge '''
        try:
            ordered_quality_matrix.copy()[edge][0]
        except KeyError:
            continue


        ''' Go through the entries of the ordered connection table  that represent the vertex to be connected with the edge '''
        for qualities_with_edges in ordered_quality_matrix.copy()[edge][0]:

             element_created=False

             target_vtx=qualities_with_edges[1]

            ''' Check first if there is sub-contour formed through the creation of an element. If there is the vertices of the sub-contour the vertices share a same id. For a connection of edge with an edge to form a non-intersecting element the id of the edge's vertices along with the target vertex must be the same. '''

            try:
                edge_key1= vertex_dict_keys[str(edge[0])]
                edge_key2= vertex_dict_keys[str(edge[1])]
                target_vertex_key=vertex_dict_keys[str(target_vtx)]
                contained_in_vertex1=False
                contained_in_vertex2=False
                for key in edge_key1:
                    if key in target_vertex_key:
                        contained_in_vertex1=True
                for key in edge_key2:
                    if key in target_vertex_key:
                        contained_in_vertex2=True
                ''' If the vertices id of the edge and the target vertex  are not the same, proceed to connect the edge with next vertex  '''
                if not contained_in_vertex1 and not contained_in_vertex2:
                    continue
            except:
                pass

            print("Edge:",edge,"targeting:",target_vtx)

            ''' If the target vtx is in set of locked vertices than proceed to connect edge with next vertex'''
            if target_vtx in set_locked_vertices:
                print(" Target vertex {} is locked".format(target_vtx))
                continue



            ''' If the element already exists proceed to next edge'''
            element=(edge[0],edge[1],target_vtx)
            print(element)
            existing_element=False
            for element in set_elements:
                if set(element)== set(element):
                    print("Element {} already in set".format(element))
                    existing_element=True
                break
            if existing_element:
                break



            set_elements.add(element)


            ''' Check if a locked vertex was created after the creation of the element
             If so, add it to the list'''
            Found_locked_vertex=False
            for vertex in element:
                if vertex<polygon.shape[0]:
                    _,isclosed = is_closed_ring(vertex,set_elements,*
                      connection_indices(vertex,get_contour_edges(polygon)))
                if isclosed and vertex not in set_locked_vertices:
                    print("Vertex locked:",vertex)
            Found_locked_vertex=True

            # New edges after creation of the element
            new_edge1=(edge[0],target_vtx)
            new_edge2=(edge[1],target_vtx)

            # Update the set of edges
            if new_edge1 not in set_edges and tuple(reversed(new_edge1)) not in
              set_edges:
                set_edges.add(new_edge1)
                interior_edges.add(new_edge1)
                print("edges inserted:",new_edge1)
                print("set of interior edges updated:",interior_edges)
                print("set of edges updated:",set_edges)
            if new_edge2 not in set_edges and tuple(reversed(new_edge2)) not in
              set_edges:
                set_edges.add(new_edge2)
                interior_edges.add(new_edge2)
                print("edges inserted:",new_edge2)
                print("set of interior edges updated:",interior_edges)
                print("set of edges updated:",set_edges)



            element_created=True

            if target_vtx>=polygon.shape[0]:
                 set_interior_edge_with_inner_point.add(new_edge1)
                 set_interior_edge_with_inner_point.add(new_edge2)

            if element_created:

                ''' Checking if locked facets are formed after the creation of the
                  element'''
                element_edges=set()
                for i in permutations(element,2):
                    element_edges.add(i)
                count=0
                indices=[]
                for index,edge_ in enumerate(list(contour_edges)):
                    if edge_ in element_edges:
                        indices.append(index)
                        count+=1
                if count==2:
                for index in indices:
                    locked_facet=list(contour_edges)[index]
                if locked_facet!= edge and locked_facet[::-1]!=edge:
                    print('spotted locked facet {}'.format(locked_facet))
                    ''' Delete the entry from connection table that includes the edge '''
                    ordered_quality_matrix.pop(locked_facet)


                set_open_vertices=set(range(len(polygon)))-set_locked_vertices
                set_interior_edge_with_inner_point_reformed=np.array(list(set_interior_edge_with_inner_point)).flatten()


                for vertex in range(len(polygon),len(polygon_with_points)):
                    if vertex not in set_interior_edge_with_inner_point_reformed:
                        set_orphan_vertices.add(vertex)


                ''' Check if sub-contours are formed after the creation of an element. If so, assign the same id to the vertices of each sub-contour'''
                sub_polygon_list=check_for_sub_polygon(set_orphan_vertices,set_open_vertices,interior_edges,set_elements,polygon,points)


                if len(sub_polygon_list)>1:
                    vertex_dict_keys=dict()
                for index,vertices in enumerate(sub_polygon_list):
                    for vertex in vertices:
                        if str(vertex) not in list((vertex_dict_keys).keys()):
                            vertex_dict_keys[str(vertex)]=[index]
                        else:
                            vertex_dict_keys[str(vertex)].append(index)

                set_orphan_vertices=set()

        break


    if plot_mesh:
        triangulated={'segment_markers': np.ones([polygon.shape[0]+points.shape[0]]), 'segments':np.array(get_contour_edges(polygon)), 'triangles': np.array(list(list(i) for i in set_elements)),
            'vertex_markers': np.ones([polygon.shape[0]+points.shape[0]]), 'vertices': np.vstack([ polygon,points])}
        plot(plt.axes(), **triangulated)
```

```python
        print("Final edges:",set_edges)
        print("Elements created:",set_elements)
        print("Set of locked vertices:", set_locked_vertices)


    # Find open vertices
    for element in set_elements:
        for vertex in element:
            if vertex>=polygon.shape[0]:
                continue
        _ ,isclosed = is_closed_ring(vertex,set_elements,*connection_indices(vertex,get_contour_edges(polygon)))
        if isclosed and vertex not in set_locked_vertices:
            print("Vertex locked:",vertex)
            Found_locked_vertex=True
            set_locked_vertices.add(vertex)
    set_open_vertices=set(range(len(polygon)))-set_locked_vertices

    # Check for vertices that are not connected to any point
    set_interior_edge_with_inner_point_reformed=np.array(list(set_interior_edge_with_inner_point)).flatten()
    for vertex in range(len(polygon),len(polygon_with_points)):
        if vertex not in set_interior_edge_with_inner_point_reformed:
            set_orphan_vertices.add(vertex)

    print("set of orphan vertex:",set_orphan_vertices)
    print("Set of open vertices:", set_open_vertices)
    set_edges.clear(),set_locked_vertices.clear(),set_forbidden_intersections.clear
    sub_element_list=[]

    ''' If there exist open vertices then the triangulation algorithm is called recursively '''
    if len(set_open_vertices)>0:

           # Get list of sub-contours
           sub_polygon_list=check_for_sub_polygon(set_orphan_vertices,set_open_vertices,interior_edges,set_elements,polygon,points)


             for sub_polygon_indices in sub_polygon_list:
                 if len(set_orphan_vertices)==0:
                    if len(sub_polygon_indices)>=3:
                     print("remeshing subpolygon",sub_polygon_indices)
                     polygon_copy=np.vstack([polygon,points])
                     sub_polygon=np.array(polygon_copy[sub_polygon_indices])

                     if not is_counterclockwise(sub_polygon):
                         sub_polygon=np.array(polygon_copy[sub_polygon_indices[::-1]])

                     sub_quality=get_quality_matrix_NN(sub_polygon,inner_points=[])
                     sub_order_matrix=Triangulation.order_quality_matrix(sub_quality,sub_polygon,check_for_equal=True)
```

```


         sub_elements,_,_=triangulate(sub_polygon,sub_order_matrix,recursive=True)
         if len(sub_elements)!=0:
             for element in sub_elements:
                indices=np.asarray(element)
                 print(element)
                 triangle=sub_polygon[indices]
                 polygon_indices=get_indices(triangle,polygon_with_points)
                 sub_element_list.append(polygon_indices)
        else:
            if len(sub_polygon_indices)>=3:


                sub_polygon_inner_points=[]
                inner_points_indices=np.asarray(list(set_orphan_vertices)).flatten()
                #inner_points_indices=np.sort(inner_points_indices)
                print("remeshing subpolygon",sub_polygon_indices)
                polygon_copy=np.vstack([polygon,points])
                sub_polygon=np.array(polygon_copy[sub_polygon_indices])


                if not is_counterclockwise(sub_polygon):
                    sub_polygon=np.array(polygon_copy[sub_polygon_indices[::-1]])

                inner_points=np.array(polygon_copy[inner_points_indices])
                inner_points=sort_points(inner_points.reshape(1,len(inner_points),2),len(inner_points)).reshape(len(inner_points),2)

                for point in inner_points:
                    is_inside=ray_tracing(point[0],point[1],sub_polygon)
                    if is_inside:
                        sub_polygon_inner_points.append(point)
                        print("Point ",point," is inside ",sub_polygon_indices)

                if len(sub_polygon_inner_points)!=0:

                    sub_polygon_inner_points=np.array(sub_polygon_inner_points)
                    sub_polygon_with_points=np.vstack([sub_polygon,sub_polygon_inner_points])
                    sub_quality=get_quality_matrix_NN(sub_polygon,sub_polygon_inner_points)
                    sub_order_matrix=order_quality_matrix(sub_quality,sub_polygon,sub_polygon_with_points,check_for_equal=True)

```

```
                                print(sub_quality,sub_order_matrix)
                                print(sub_polygon)
                                sub_elements,_=triangulate(sub_polygon,sub_polygon_inner_points,sub_order_matrix,recursive=True)
                                if len(sub_elements)!=0:
                                    for element in sub_elements:
                                        indices=np.asarray(element)
                                        print(element)
                                        triangle=sub_polygon_with_points[indices]
                                        polygon_indices=get_indices(triangle,polygon_with_points)
                                        sub_element_list.append(polygon_indices)
                            else:
                                 sub_quality=get_quality_matrix_NN(sub_polygon,inner_points=[])
                                sub_order_matrix=Triangulation.order_quality_matrix(sub_quality,sub_polygon,check_for_equal=True)

                                print(sub_quality,sub_order_matrix)
                                sub_elements,_=triangulate(sub_polygon,inner_points=[],sub_order_matrix,recursive=True)
                                if len(sub_elements)!=0:
                                    for element in sub_elements:
                                        indices=np.asarray(element)
                                        print(element)
                                        triangle=sub_polygon[indices]
                                        polygon_indices=get_indices(triangle,polygon_with_points)
                                        sub_element_list.append(polygon_indices)

    return set_elements,sub_element_list
```

B.2 Mesh Improvement

B.2.1 Mesh class

```
    ''' The class creates mesh objects reading .vtk files. All mesh improvement operation are functions of the present class '''
    class ModifiableMesh(meshio.Mesh):
        def __init__(self, points, cells, point_data=None, cell_data=None, field_data=None, point_sets=None, cell_sets=None, gmsh_periodic=None, info=None, normal=None, target_edgelength_boundary=None, target_edgelength_interface=None):
        super().__init__(points, cells, point_data, cell_data, field_data, point_sets, cell_sets, gmsh_periodic, info)

        self.vertex_index = None
        self.line_index = None
        self.triangle_index = None
        self.tetra_index = None
        for c, cell in enumerate(self.cells):
            if cell.type == 'vertex':
                self.vertex_index = c
            elif cell.type == 'line':
                self.line_index = c
            elif cell.type == 'triangle':
                self.triangle_index = c
            elif cell.type == 'tetra':
                self.tetra_index = c

        if self.tetra_index is not None:
            self.dimension = 3
        elif self.triangle_index is not None:
            self.dimension = 2
        else:
            self.dimension = 1

        try:
            self.fixed_vertices = self.get_vertices().reshape(-1)
        except TypeError:
            self.fixed_vertices = np.array([], dtype=np.int)

        boundary = np.zeros(self.points.shape[0], dtype=np.bool)
            try:
            for object in self.get_lines():
                for vertex in object:
                    if vertex not in self.fixed_vertices:
                        boundary[vertex] = True
            except TypeError:
                pass

        interior = np.zeros(self.points.shape[0], dtype=np.bool)
        for vertex in range(len(self.points)):
            objects = self.get_neighbourhood(vertex)
            try:
                _, index, _ = self.get_contour(objects)
            except:
                continue
            interior[vertex] = vertex not in index

        self.interior_vertices = np.setdiff1d(np.nonzero(~boundary & interior)[0], self.fixed_vertices)
        self.boundary_vertices = np.setdiff1d(np.nonzero(boundary & ~interior)[0], self.fixed_vertices)
        self.interface_vertices = np.setdiff1d(np.nonzero(boundary & interior)[0], self.fixed_vertices)

        if normal is None:
            normal = [0,0,1]
        self.normal = np.array(normal)
        if target_edgelength_boundary is None:
            edges = np.copy(self.get_lines())
            valid = np.concatenate([self.boundary_vertices, self.fixed_vertices])
            edges = edges[np.isin(self.get_lines()[:,0], valid) & np.isin(self.get_lines()[:,1], valid)]
            length = np.linalg.norm(self.points[edges[:,0]] - self.points[edges[:,1]], axis=1)
            target_edgelength_boundary = np.mean(length)
        self.target_edgelength_boundary = target_edgelength_boundary

        if target_edgelength_interface is None:
        try:
            edges = np.copy(self.get_lines())
            valid = np.concatenate([self.interface_vertices, self.fixed_vertices])
            edges = edges[np.isin(self.get_lines()[:,0], valid) & np.isin(self.get_lines()[:,1], valid)]
            length = np.linalg.norm(self.points[edges[:,0]] - self.points[edges[:,1]], axis=1)
            target_edgelength_interface = np.mean(length)
        except:
            target_edgelength_interface = target_edgelength_boundary
            self.target_edgelength_interface = target_edgelength_interface

        self.refinement_threshold =4/3
        self.coarsen_threshold = 4/5


        self.generator = np.random.Generator(np.random.PCG64())

    def get_vertices(self):
        if self.vertex_index is None:
            return None
        else:
            return self.cells[self.vertex_index].data

    def get_lines(self):
        if self.line_index is None:
            return None
        else:
            return self.cells[self.line_index].data

    def set_lines(self, lines):
        self.cells[self.line_index] = self.cells[self.line_index]._replace(data=lines)

    def get_triangles(self):
        if self.triangle_index is None:
            return None
        else:
            return self.cells[self.triangle_index].data

    def set_triangles(self, triangles):
        self.cells[self.triangle_index] = self.cells[self.triangle_index]._replace(data=triangles)

    def get_tetras(self):
        if self.tetra_index is None:
            return None
        else:
            return self.cells[self.tetra_index].data

    def set_tetras(self, tetras):
    self.cells[self.tetra_index] = self.cells[self.tetra_index]._replace(data=tetra)

    def get_elements(self):
        if self.dimension == 3:
            return self.get_tetras()
        elif self.dimension == 2:
            return self.get_triangles()

    def set_elements(self, elements):
        if self.dimension == 3:
            self.set_tetras(elements)
        elif self.dimension == 2:
            self.set_triangles(elements)


'''Calculates the quality of each element in the mesh.'''
    def quality(self):

        if self.dimension == 2:
            quality = np.apply_along_axis(self.triangle_quality, 1, self.get_triangles())
        elif self.dimension == 3:
            quality = self.to_pyvista().quality
        else:
            raise DimensionError('Mesh must be a surface or volume mesh')
        return quality
```

B.2.2 Reconnection

```



''' Function calling neural networks to retriangulate a contour '''
@torch.no_grad()
def retriangulate(contour):
    if len(contour) == 3:
        return np.array([0,1,2], dtype=np.int)

    net = get_connectivity_network(contour.shape[0])

    procrustes = apply_procrustes(contour)
    input = torch.tensor(np.asarray(procrustes, dtype=np.float32).reshape((1,-1)))
    table = net(input)

    table = table[0].numpy().reshape([contour.shape[0], contour.shape[0]])



    index = np.arange(len(contour))
    ordered_triangles = order_triangles(contour, table)

    chosen, cavities = triangulate2(procrustes, index, ordered_triangles)
    new_elements = ordered_triangles[chosen]

    while len(cavities) > 0:
        cavity = cavities.pop()
        sub = contour[cavity]

        if len(cavity) == 3:
        new_elements = np.append(new_elements, np.array([cavity]), axis=0)
        else:
        net = get_connectivity_network(sub.shape[0])

        sub_procrustes = apply_procrustes(sub)
        input = torch.tensor(sub_procrustes.astype(np.float32)).reshape((1,-1))
        table = net(input)

        table = table[0].numpy().reshape([sub.shape[0], sub.shape[0]])

        ordered_triangles = order_triangles(sub, table)

        chosen, sub_cavities = triangulate2(sub_procrustes, np.arange(len(contour)), ordered_triangles)
        new_elements = np.append(new_elements, np.take(cavity, ordered_triangles[chosen]), axis=0)
        cavities += np.take(cavity, sub_cavities)



    return new_elements





'''
Reconnect the vertices inside a cavity given by objects using a neural network.
'''
def reconnect_objects(self, objects):

    quality = np.apply_along_axis(self.triangle_quality, 1, self.get_triangles()[objects])
    q = np.min(quality)
    contour, index, _ = self.get_contour(objects)

    if len(index) > 10:
        return False, objects

    contour = contour[:-1,:2] # 2D only !
    index = index[:-1]
    rolled = np.roll(contour, 1, axis=0)
    contour_direction = np.sign(np.sum(contour[:,1]*rolled[:,0] - contour[:,0]*rolled[:,1]))
    if contour_direction < 0:
        contour = contour[::-1]
        index = index[::-1]

    if len(index) == 3:
        return True, index[None,:]

    new = retriangulate(contour)

    new_elements = np.take(index, new)
    new_quality = np.apply_along_axis(self.triangle_quality, 1, new_elements)
    if np.min(new_quality) > q:
        accepted = True
    else:
        accepted = False

    return accepted, new_elements




''' Reconnect the mesh using neural networks. Iterated until no further improvement is made. '''
def reconnect(self, maxiter=10):

    accepted = 1
    iter = 1
    while accepted > 0 and iter <= maxiter:
        try:
            partition = self.connectivity_partition()
        except:
            accepted=0
            break


        if len(partition) > 0:
            groups = np.unique(partition)
            groups = groups[groups >= 0]
            keep_elements = np.ones(len(self.get_triangles()), dtype=np.bool)
            new_elements = []
            accepted = 0
            for i, g in enumerate(groups):
                objects = partition == g
                if np.count_nonzero(objects) > 1:
                    accept, new = self.reconnect_objects(objects)
                if accept:
                    keep_elements = np.logical_and(~objects, keep_elements)
                    try:
                        new_elements = np.append(new_elements, new, axis=0)
                    except:
                        new_elements = new
            accepted += 1

            elements = self.get_triangles()[keep_elements]

            if len(new_elements) > 0:
            elements = np.append(elements, new_elements, axis=0)
            self.set_triangles(elements)
            print('Quality after {} reconnecting iterations: {}'.format(iter, np.min(self.quality())))
            iter += 1
        else:
            accepted = 0
```

B.2.3 Vertex repositioning

```

''' Return the location of a repositioned vertex '''
@torch.no_grad()
def smooth_interior_point(contour):
    net = get_smoothing_network(contour.shape[0])
    procrustes_transform, inverse_transform, _ = get_procrustes_transform(contour)
    procrustes = procrustes_transform(contour)
    shape = procrustes.reshape(-1)
    shape = np.asarray(shape, dtype=np.float32)
    input = torch.from_numpy(shape)
    prediction = net(input[None,:])[0]
    return inverse_transform(prediction.numpy())

'''
Reposition a vertex.
'''
def smooth_vertex(self, vertex):

    objects = self.get_neighbourhood(vertex)
    try:
        contour, index, _ = self.get_contour(objects)
    except:
        accepted=False
        return accepted
    if len(contour) > 10:
        accepted = False
    else:
        quality = np.apply_along_axis(self.triangle_quality, 1, self.get_triangles()[objects])
        q = np.min(quality)
        old_point = np.copy(self.points[vertex])
        contour = contour[:-1,:2] # 2D only !
        new_point = smooth_interior_point(contour) #, len(interior)
        self.points[vertex][:2] = new_point # 2D only !
        quality = np.apply_along_axis(self.triangle_quality, 1, self.get_triangles()[objects])
        accepted = True
        if np.min(quality) <= q:
            self.points[vertex] = old_point
            accepted = False
    return accepted




'''
Returns an ordered list of interior vertices to be repositioned.
'''
def smoothing_partition(self):

    quality = self.quality()
    vertex_quality = np.zeros(self.interior_vertices.shape)
        for v, vertex in enumerate(self.interior_vertices):
        objects = objects_boundary_includes(self.get_triangles(), vertex)
    vertex_quality[v] = np.min(quality[objects])
    partition = self.interior_vertices[vertex_quality < 0.8]
    vertex_quality = vertex_quality[vertex_quality < 0.8]
    if len(vertex_quality) > 0:
        partition = partition[np.argsort(vertex_quality)]
    return partition



'''
Apply vertex repositioning to a mesh using neural networks. Iterates until no further improvement is made.
'''
def smooth(self, maxiter=10):

    accepted = 1
    iter = 1
    while accepted > 0 and iter <= maxiter:
        try:
            partition = self.smoothing_partition()
        except:
            accepted=0
            break
        if len(partition) > 0:
            accepted = 0
            for v in partition:
                if self.smooth_vertex(v):
                    accepted += 1
            print('Quality after {} smoothing iterations: {}'.format(iter, np.min(self.quality())))
            iter += 1
        else:
            accepted = 0
```

B.2.4 Boundary/interface vertex repositioning

```


'''
Apply vertex repositioning a boundary vertex using NN.
'''
@torch.no_grad()
def smooth_boundary_point(contour, tangents):
    net = get_boundary_network(contour.shape[0]-1)
    procrustes_transform, inverse_transform, tangent_transform = get_procrustes_transform(contour)
    procrustes = procrustes_transform(contour)
    shape = np.concatenate([procrustes.reshape(-1), tangent_transform(tangents).reshape(-1)])
    shape = np.asarray(shape, dtype=np.float32)
    input = torch.from_numpy(shape)
    prediction = net(input[None,:])[0]

    return inverse_transform(prediction.numpy())



'''
Apply vertex repositioning a boundary vertex.
'''
def smooth_boundary_vertex(self, vertex):

    objects = self.get_neighbourhood(vertex)
    if np.count_nonzero(objects) == 1:
    return False

    contour, index = self.get_open_contour(objects, vertex)
    if len(contour) < 3:
        return False
    if len(contour) > 6:
        return False
    contour = contour[:,:2] # 2D only !

    old_point = np.copy(self.points[vertex])
    quality = np.apply_along_axis(self.triangle_quality, 1, self.get_triangles()[objects])
    q = np.min(quality)

    try:
        spline, derivative = self.get_spline([index[0], vertex, index[-1]])
    except:
        self.points[vertex] = old_point
        accepted = False
        return accepted

    tangents = derivative(np.array([0,1]))
    tangents /= np.linalg.norm(tangents, axis=1)[:,None]

    new_point = smooth_boundary_point(contour, tangents)

    fun = lambda s: np.dot(new_point - spline(s), derivative(s))
    try:
        s0 = brentq(fun, 0, 1)
        new_point = spline(s0)
        self.points[vertex][:2] = new_point # 2D only !
        quality = np.apply_along_axis(self.triangle_quality, 1, self.get_triangles()[objects])
        accepted = np.min(quality) > q
    except ValueError:
        accepted = False

    if not accepted:
        self.points[vertex] = old_point

    return accepted


'''
Returns an ordered list of boundary vertices to apply vertex repositioning.
'''
def boundary_partition(self):

    quality = self.quality()
    vertex_quality = np.zeros(self.boundary_vertices.shape)
    for v, vertex in enumerate(self.boundary_vertices):
        objects = objects_boundary_includes(self.get_triangles(), vertex)
        vertex_quality[v] = np.min(quality[objects])
    partition = self.boundary_vertices[vertex_quality < 0.8]
    vertex_quality = vertex_quality[vertex_quality < 0.8]
    if len(vertex_quality) > 0:
        partition = partition[np.argsort(vertex_quality)]
    return partition



'''
Apply vertex repositioning to the boundary vertices of a mesh using neural networks. Iterates until no further improvement is made.
```

```
'''

  def smooth_boundary(self, maxiter=10):

    accepted = 1
    iter = 1
    while accepted > 0 and iter <= maxiter:
        partition = self.boundary_partition()
        if len(partition) > 0:
            accepted = 0
            for vertex in partition:
                if self.smooth_boundary_vertex(vertex):
                    accepted += 1
            print('Quality after {} boundary smoothing iterations: {}'.format(iter, np.min(self.quality())))
            iter += 1
        else:
            accepted = 0


'''
Apply vertex repositioning to a interface vertex using NNs.
'''
@torch.no_grad()
def smooth_interface_point(contour, points, tangents):
    net = get_interface_network(contour.shape[0])
    procrustes_transform, inverse_transform, tangent_transform = get_procrustes_transform(contour)
    procrustes = procrustes_transform(contour)
    interface = procrustes_transform(points)
    shape = np.concatenate([procrustes.reshape(-1), interface.reshape(-1), tangent_transform(tangents).reshape(-1)])
    shape = np.asarray(shape, dtype=np.float32)
    input = torch.from_numpy(shape)
    prediction = net(input[None,:])[0]
    return inverse_transform(prediction.numpy())




'''
Returns an ordered list of interface vertices to be apply vertex reposition.
'''
def interface_partition(self):

    quality = self.quality()
    vertex_quality = np.zeros(self.interface_vertices.shape)
    for v, vertex in enumerate(self.interface_vertices):
        objects = objects_boundary_includes(self.get_triangles(), vertex)
        vertex_quality[v] = np.min(quality[objects])
    partition = self.interface_vertices[vertex_quality < 0.9] #np.mean(quality)
    vertex_quality = vertex_quality[vertex_quality < 0.9]
    if len(vertex_quality) > 0:
        partition = partition[np.argsort(vertex_quality)]
    return partition



'''
Apply vertex repositioning to a interface vertex.
'''

def smooth_interface_vertex(self, vertex):

    objects = self.get_neighbourhood(vertex)
    if np.count_nonzero(objects) == 1:
        return False
    old_point = np.copy(self.points[vertex])

    try:
        contour, index, _ = self.get_contour(objects)
    except:
        self.points[vertex] = old_point
        accepted = False
        return accepted

    contour = contour[:,:2] # 2D only !

    quality = np.apply_along_axis(self.triangle_quality, 1, self.get_triangles()[
      objects])
    q = np.min(quality)

    interface = np.intersect1d(index, np.union1d(self.interface_vertices, self.
      fixed_vertices))

    try:
        spline, derivative = self.get_spline([interface[0], vertex, interface[-1]])
    except:
        self.points[vertex] = old_point
        accepted=False
        return accepted

    tangents = derivative(np.array([0,1]))
    tangents /= np.linalg.norm(tangents, axis=1)[:,None]

    try:
        new_point = smooth_interface_point(contour, self.points[interface,:2], tangents
          )
    except:
        self.points[vertex] = old_point
        accepted=False
        return accepted

    fun = lambda s: np.dot(new_point - spline(s), derivative(s))
    try:
        s0 = brentq(fun, 0, 1)
        new_point = spline(s0)
        self.points[vertex][:2] = new_point # 2D only !
        quality = np.apply_along_axis(self.triangle_quality, 1, self.get_triangles()[objects])
        accepted = np.min(quality) > q
    except ValueError:
        accepted = False

    if not accepted:
        self.points[vertex] = old_point

    return accepted



'''
Apply vertex repositioning to the interface vertices of a mesh using neural networks. Iterates until no further improvement is made.
'''
def smooth_interface(self, maxiter=10):

    accepted = 1
    iter = 1
    while accepted > 0 and iter <= maxiter:
        partition = self.interface_partition()
        if len(partition) > 0:
            accepted = 0
            for vertex in partition:
                if self.smooth_interface_vertex(vertex):
                    accepted += 1
            print('Quality after {} interface smoothing iterations: {}'.format(iter, np.min(self.quality())))
            iter += 1
        else:
            accepted = 0
```

B.2.5 Edge Length control

```

'''Triangulate a cavity with inner points using NNs'''

@torch.no_grad()
def retriangulate_with_interior(contour, *args):
    try:
        net = get_connectivity_network(contour.shape[0], len(args))
    except FileNotFoundError:
        if len(args) == 1:
        return simple_retriangulate(contour, *args)
    else:
        raise ValueError('Network for {} edges and {} interior points not trained'.
          format(contour.shape[0], len(args)))

    procrustes_transform,_,_ = get_procrustes_transform(contour)
    procrustes = procrustes_transform(contour)
    inner = procrustes_transform(np.array(args))
    input = torch.tensor(np.concatenate([np.asarray(np.append(procrustes, procrustes
      [0][None,:], axis=0), dtype=np.float32), np.asarray(inner, dtype=np.float32)],
      axis=0)[None,None,:,:])


    table = net(input)
    table = table[0].numpy().reshape([contour.shape[0], contour.shape[0]+len(args)])


    ordered_matrix = order_quality_matrix(table, procrustes, np.vstack([procrustes,
      inner]), check_for_equal=False)
    new_elements, sub_elements = triangulate(procrustes, inner, ordered_matrix,
      recursive=True, plot_mesh=False)



    new_elements = list(new_elements)


    return np.array(new_elements + sub_elements, dtype=np.int)




'''
Refine a cavity given by objects using a neural network.
'''
def refine_objects(self, objects, new_points):

    new_index = np.arange(len(self.points), len(self.points) + len(new_points))
    self.points = np.append(self.points, new_points, axis=0)
    self.interior_vertices = np.append(self.interior_vertices, new_index)


    try:
        contour, index, interior = self.get_contour(objects)
    except ValueError:
        print('Invalid contour')
        return False, None, None

    contour = contour[:-1,:2] # 2D only !
    index = index[:-1]
    rolled = np.roll(contour, 1, axis=0)
    contour_direction = np.sign(np.sum(contour[:,1]*rolled[:,0] - contour[:,0]*rolled
      [:,1]))
    if contour_direction < 0:
            contour = contour[::-1]
            index = index[::-1]
        index = np.append(index, new_index)

        new = retriangulate_with_interior(contour, *new_points[:,:2])
        new_elements = np.take(index, new)

        new_quality = np.apply_along_axis(self.triangle_quality, 1, new_elements)
        if np.min(new_quality) > 0:
            accepted = True
        else:
            accepted = False
            self.points = self.points[:-len(new_points)]
            self.interior_vertices = self.interior_vertices[:-len(new_points)]

        return accepted, new_elements, interior




    '''
    Partition the mesh into cavities to be refined.
    '''
    def refinement_partition(self):

        partition = np.arange(len(self.get_triangles()))

        elements = self.get_triangles()
        all_edges = [np.sort(np.roll(e, r)[:2]) for r in range(3) for e in elements]
        edges, counts = np.unique(all_edges, axis=0, return_counts=True)

        is_interior = counts > 1
        edges = edges[is_interior]

        if len(self.interface_vertices > 0):
            is_interface = objects_boundary_includes_some(edges, 2, *self.interface_vertices)
            edges = edges[~is_interface]

        length = np.linalg.norm(self.points[edges[:,0]] - self.points[edges[:,1]], axis=1)
        long = length > self.target_edgelengths(edges) * self.refinement_threshold
        edges = edges[long]
        edges = edges[np.argsort(-length[long])]

        new_points = {}
        not_accepted = []
        for edge in edges:
            triangle_pair = self.find_triangles_with_common_edge(edge)
            group = np.min(partition[triangle_pair])
            other_group = np.max(partition[triangle_pair])

            first = partition == group
        second = partition == other_group
        partition[np.logical_or(first, second)] = group

        accept_group = True
        if group not in new_points and other_group not in new_points:
            new_points[group] = np.array([(self.points[edge[0]] + self.points[edge[1]]) / 2])
        else:
            new_polygon_objects = partition == group
            contour, _, interior = self.get_contour(new_polygon_objects)
            nodes = [new_points[g] for g in [group, other_group] if g in new_points]
            new = sum([len(n) for n in nodes])
            if len(contour) > 8 or len(interior) + new > len(contour) - 4:
                accept_group = False
            else:
                new_points[group] = np.concatenate(nodes + [np.array([(self.points[edge[0]] + self.points[edge[1]]) / 2])], axis=0)
                if other_group in new_points:
                del new_points[other_group]

    if not accept_group:
        partition[second] = other_group

    partition[np.isin(partition, list(new_points.keys()), invert=True)] = -1

    return partition, new_points






'''
Refine (regulate long edges) the mesh using neural networks. Iterated until no further improvement is made.
'''
def refine(self, maxiter=10):

    accepted = 1
    iter = 1
    while accepted > 0 and iter <= maxiter:
        partition, new_points = self.refinement_partition()
        groups = new_points.keys()
        if len(groups) > 1:
        keep_elements = np.ones(len(self.get_triangles()), dtype=np.bool)
        new_elements = []
        accepted = 0
        for g in groups:
            if g >= 0:
                objects = partition == g
                if np.count_nonzero(objects) > 1:
                accept, new, interior = self.refine_objects(objects, new_points[g])
                if len(interior) > 0:
                    raise ValueError('Points to be deleted during refinement')
                if accept:
                    keep_elements = keep_elements & ~objects
                    try:
                        new_elements = np.append(new_elements, new, axis=0)
                    except:
                        new_elements = new
                    accepted += 1

            if len(new_elements) > 0:
                elements = self.get_triangles()[keep_elements]
                elements = np.append(elements, new_elements, axis=0)
                self.set_triangles(elements)
                print('Quality after {} refinement iterations: {}'.format(iter, np.min(self.quality())))
                iter += 1
            else:
                accepted = 0
        else:
        accepted = 0


'''
Partition the mesh into cavities to be coarsened.
'''
def coarsen_partition(self):

    partition = np.arange(len(self.get_triangles()))

    elements = self.get_triangles()
    all_edges = [np.sort(np.roll(e, r)[:2]) for r in range(3) for e in elements]
    edges = np.unique(all_edges, axis=0)

    boundary_or_interface = np.concatenate([self.boundary_vertices, self.interface_vertices])
    includes_boundary = objects_boundary_includes_some(edges, 1, *boundary_or_interface)
    edges = edges[~includes_boundary]


    length = np.linalg.norm(self.points[edges[:,0]] - self.points[edges[:,1]], axis=1)
    short = length < self.target_edgelengths(edges) * self.coarsen_threshold
    edges = edges[short]
    edges = edges[np.argsort(length[short])]

    new_points = {}
    not_accepted = []
    for edge in edges:
        potential = objects_boundary_includes_some(self.get_triangles(), 1, *edge)
        group = np.min(partition[potential])
        all_groups = np.unique(partition[potential])
        selected = np.isin(partition, all_groups)

        if np.all(np.isin(all_groups, list(new_points.keys()), invert=True)):
            partition[selected] = group
            new_points[group] = np.array([(self.points[edge[0]] + self.points[edge[1]]) / 2])
        else:
            try:
                contour, _, interior = self.get_contour(selected)
            except:
                continue
            interior = interior[np.isin(interior, edge, invert=True)]
            nodes = [new_points[g] for g in all_groups if g in new_points]
            new = sum([len(n) for n in nodes])
            if len(contour) < 10 and len(interior) + new < len(contour) - 3:
                partition[selected] = group
                new_points[group] = np.concatenate(nodes + [np.array([(self.points[edge[0]] + self.points[edge[1]]) / 2])], axis=0)
                for g in all_groups:
                    if g != group and g in new_points:
                        del new_points[g]

    partition[np.isin(partition, list(new_points.keys()), invert=True)] = -1

    return partition, new_points



'''
Coarsen the mesh using neural networks. Iterated until no further improvement is made.
'''
def coarsen(self, maxiter=10):

    accepted = 1
    iter = 1
    while accepted > 0 and iter <= maxiter:
    self.coarsen_near_boundary_or_interface()
    partition, new_points = self.coarsen_partition()
    groups = new_points.keys()
    if len(groups) > 1:
        keep_elements = np.ones(len(self.get_triangles()), dtype=np.bool)
        new_elements = []
        accepted = 0
        for g in groups:
            if g >= 0:
                objects = partition == g
                if np.count_nonzero(objects) > 1:
                    accept, new, remove = self.refine_objects(objects, new_points[g])
                    if accept:
                        keep_elements = np.logical_and(~objects, keep_elements)
                        try:
                            new_elements = np.append(new_elements, new, axis=0)
                        except:
                            new_elements = new
                self.points = np.delete(self.points, remove, axis=0)
                remains = np.isin(self.interior_vertices, remove, invert=True)
                self.interior_vertices = self.interior_vertices[remains]
                for old in remove:
                    remove[remove > old] -= 1
                    new_elements[new_elements > old] -= 1
                    self.interior_vertices[self.interior_vertices > old] -= 1
                    self.interface_vertices[self.interface_vertices > old] -= 1
                    self.boundary_vertices[self.boundary_vertices > old] -= 1
                    self.fixed_vertices[self.fixed_vertices > old] -= 1
                    for cell in self.cells:
                    cell.data[cell.data > old] -= 1
                accepted += 1

        if len(new_elements) > 0:
            elements = self.get_triangles()[keep_elements]
            elements = np.append(elements, new_elements, axis=0)
            self.set_triangles(elements)
            print('Quality after {} coarsening iterations: {}'.format(iter, np.min(self.quality())))
            iter += 1
        else:
            accepted = 0
    else:
        accepted = 0
```

B.3 Large Mesh generation

```
''' shapely module is used for projecting point of the low resolution contour to the high resolution contour '''
import shapely
import shapely.geometry as geom
from shapely.geometry import MultiPoint
from shapely.ops import nearest_points
import torch

''' Function inserting equidistance points to an edge according to a uniform
     target edge length. '''

def insert_point_edges_uniform(edge,target_length):
    edge=edge.astype('float64')
    edge_length=np.linalg.norm(edge[0]-edge[-1],2)

    ratio=int(edge_length/target_length)
    k=ratio
    x1,x2=edge[0][0],edge[1][0]
    k=1

    if ratio>4:
        k=5
    if ratio<=1:
        return edge

    x1,y1=edge[0][0],edge[0][1]
    x2,y2=edge[1][0],edge[1][1]
    for i in range(1,k):
        point=np.array([[x1+(i/k)*(x2-x1),y1+(i/k)*(y2-y1)]])
        edge=np.insert(edge,-1,point,axis=0)
    return edge


''' Function inserting equidistance points to an edge according to a size function.'''
def insert_point_edges_adaptive(edge,size_function):
    epsilon=1e-1
    original_edge=edge.astype('float64')
    k=1
    continue_dividing=True
    x1,y1=edge[0][0],edge[0][1]
    x2,y2=edge[1][0],edge[1][1]
    x3,y3=0.5*(x1+x2),0.5*(y1+y2)
    edge_length=np.linalg.norm(edge[0]-edge[1],2)
    ratio1=int(edge_length/size_function(x1,y1))
    ratio2=int(edge_length/size_function(x2,y2))
    if ratio1<=1 and ratio2<=1:
        return edge
    while(continue_dividing):
        k+=1
        edge=original_edge

        for i in range(1,k):
            point=np.array([[x1+(i/k)*(x2-x1),y1+(i/k)*(y2-y1)]])
            edge=np.insert(edge,-1,point,axis=0)
        edge_length=np.linalg.norm(edge[0]-edge[1],2)
        if k==5 or np.linalg.norm(edge_length-size_function(edge[1][0],edge[1][1]))<epsilon:
            continue_dividing=False

    return edge



''' Function determining if point of the low resolution is a point of the high resolution contour.'''
def is_on_surface(point,surface_points):
    point=geom.Point(point[0],point[1])
    return surface_points.contains(point)


''' Function returning edge lengths of a contour.'''
def get_edge_lengths(polygon):
    polygon_edge_lengths=np.empty([polygon.shape[0]])
    for index,_ in enumerate(polygon):
        polygon_edge_lengths[index]=np.linalg.norm(polygon[(index+1)%(polygon.shape[0])]-polygon[index])
    return polygon_edge_lengths



''' Function performing procrustes transformation (includes inverse transformation)'''
def get_procrustes_transform(polygon):
    centralised_ref_polygon, norm_ss_ref_polygon = get_reference_data(polygon.shape[0])

    mu_polygon = polygon.mean(0)
    centralised_polygon = polygon - mu_polygon
    ss_polygon = (centralised_polygon**2).sum()
    norm_ss_polygon = np.sqrt(ss_polygon)
    centralised_polygon /= norm_ss_polygon



    A = np.dot(centralised_ref_polygon.T, centralised_polygon)
    U, s, Vt = np.linalg.svd(A, full_matrices=False)
    V= Vt.T
    R = np.dot(V, U.T)
    traceTA = s.sum()
    Rinv = R.T

    scaling_factor= norm_ss_ref_polygon * traceTA


    def procrustes_transform(polygon):
        return norm_ss_ref_polygon * traceTA * np.dot((polygon - mu_polygon) / norm_ss_polygon, R)

    def inverse_transform(polygon):
        return np.dot(polygon, Rinv) / (norm_ss_ref_polygon * traceTA) * norm_ss_polygon + mu_polygon

    def tangent_transform(tangents):
        return np.dot(tangents, R)

    return procrustes_transform, inverse_transform, tangent_transform, scaling_factor


''' Function calling NN1 to predict number of inner points'''
@torch.no_grad()
def get_nb_of_points(polygon,target_edge_length):

    nb_of_edges=len(polygon)

    # Load trained network for prediction of number of points
```

```python
    with open('../network_datasets/number_of_points_NN/'+str(nb_of_edges)+'_NN_nb_of_points.pkl','rb') as f:
        NN1=pickle.load(f)

    procrustes,_,_ ,scaling_factor= get_procrustes_transform(polygon)

    target_edge_length*=scaling_factor

    input = torch.tensor(np.concatenate([np.asarray((procrustes(polygon), axis=0), dtype=np.float32),np.asarray((target_edge_length, axis=0), dtype=np.float32)]))
    nb_of_points= NN1(input)
    nb_of_points=int(np.round(nb_of_points))

    return nb_of_points


''' Function calling NN2 to predict number of inner points '''
def get_inner_vertices(polygon,target_edge_length,nb_of_points):

    # Define the grid
    nb_of_grid_points=20

    X=np.linspace(-1.2,1.2,nb_of_grid_points)
    Y=np.linspace(-1.2,1.2,nb_of_grid_points)
    XX,YY=np.meshgrid(X,Y)
    grid_points=np.array([[x,y] for x in X for y in Y])

    nb_sectors=int(nb_of_grid_points/2)
    sectors,indices=seperate_to_sectors(grid_points,nb_sectors,nb_of_grid_points)
    grid_step_size=int(nb_of_grid_points/nb_sectors)

    network_filepath=str(nb_of_edges)+'_polygons/'+str(nb_of_edges)+'_'+str(nb_of_points)+'_polygons/networks/'+str(nb_of_edges)+'_'+str(nb_of_points)+'_'+str(nb_of_grid_points)+'_grid_NN'


    # load network
    with open(network_filepath,'rb') as f:
        NN2=pickle.load(f)


    procrustes,inverse,_ ,scaling_factor= get_procrustes_transform(polygon)

    polygon=procrustes(polygon)

    target_edge_length*=scaling_factor

    # use network to extract predicted points
    polygon_with_target_edge_length=np.hstack([polygon.reshape(2*(len(polygon))),np.array(target_edge_length).reshape(1)])

    # Adding grid points of each patch for the input of the NN
    polygon_with_grid_points=[]
    for sector in sectors:
        polygon_with_sector_points=np.hstack([polygon_with_target_edge_length.reshape(1,len(polygon_with_target_edge_length)),sector.reshape(1,2*len(sector))])
        polygon_with_sector_points=Variable(torch.from_numpy(polygon_with_sector_points))
        polygon_with_sector_points=polygon_with_sector_points.expand(1,polygon_with_sector_points.shape[1]).type(torch.FloatTensor)
        polygon_with_grid_points.append(polygon_with_sector_points)



    # Infer grid scores ftom NN
    sector_qualities=[]
    for polygon_with_sector_points in polygon_with_grid_points:
        sector_quality=NN2(polygon_with_sector_points)
        sector_qualities.append(sector_quality.data[0].numpy())

    sector_qualities=np.array(sector_qualities)


    grid_qualities=np.empty((grid_step_size**2)*(nb_sectors**2))
    for index,point_index in enumerate(indices):
    grid_qualities[point_index]=sector_qualities.flatten()[index]


    # Point selection
    predicted_points,surrounding_points_list,grid_qualities_surrounding=select_points(polygon,grid_points,grid_qualities,nb_of_points,nb_of_grid_points,target_edge_length)


    # Interpolate
    predicted_points=[point  for i in range(nb_of_points) for point in bilineaire_interpolation(surrounding_points_list[i],grid_qualities_surrounding[i],predicted_points[i])]
    predicted_points=np.array(predicted_points).reshape(nb_of_points,2)
    predicted_points=np.unique(predicted_points,axis=0)


    predicted_points=inverse(predicted_points)

    return predicted_points


''' Function calling NN3 to mesh a subcontour with inner inserted points'''
@torch.no_grad()
def triangulate_NN(polygon, inner_points):

    # Load trained connectivity network
    with open('../network_datasets/connectivity_NN/'+str(len(polygon))+'_'+str(len(inner_points))+'_NN_qualities.pkl','rb') as f:
        NN3=pickle.load(f)



    procrustes_transform,_,_ = get_procrustes_transform(polygon)
    procrustes = procrustes_transform(polygon)
    inner_points = procrustes_transform(np.array(inner_points))
    input = torch.tensor(np.concatenate([np.asarray(np.append(procrustes, procrustes[0][None,:], axis=0), dtype=np.float32), np.asarray(inner_points, dtype=np.float32)], axis=0)[None,None,:,:])

    table = NN3(input)

    table = table[0].numpy().reshape([polygon.shape[0], polygon.shape[0]+len(args)])


    ordered_matrix = order_quality_matrix(table, procrustes, np.concatenate([procrustes, inner], axis=0))
    try:
        new_elements, sub_elements = triangulate(procrustes, inner, ordered_matrix, recursive=True)
    except RuntimeError:
        return np.array([])

    return np.array(list(new_elements) + sub_elements, dtype=np.int)


''' Function that calls vertex repositioning(smooth) during large mesh generation '''
def smooth_recursive(sub_contours,elements_lists):
    sub_contours=np.array(sub_contours)
    sub_contours_reshaped=sub_contours.reshape(-1,2)
    points_in_mesh,inverse_indices=np.unique(sub_contours_reshaped,axis=0,return_inverse=True)


    elements_indices=[]
    for sub_contour,elements_list in zip(sub_contours,elements_lists):

        for element in elements_list:
            element_global_index=[]

            element_coordinates=sub_contour[element]
            signed_area=compute_triangle_area(element_coordinates)
            if signed_area==0:
                element_coordinates[1],element_coordinates[2]=element_coordinates[2],element_coordinates[1]
                print("FOUND NEGATIVE")


            for coordinate in element_coordinates:
                for index,point in enumerate(points_in_mesh):
                    if np.allclose(coordinate,point):
                        element_global_index.append(index)
            if len(element_global_index) ==len(set(element_global_index)):
                elements_indices.append(element_global_index)
    elements_indices=np.array(elements_indices)
    cells=dict({"triangle":elements_indices})
    mesh=meshio.Mesh(points_in_mesh,cells)
    meshio.write('original.vtk',mesh)
    mesh = mymesh.read('original.vtk')
    mesh.smooth()
    new_points=mesh.points[:,:2]
    new_subcontours=new_points[inverse_indices].reshape(len(sub_contours),6,2)
    return new_subcontours





''' Starting from a low resolution contour (low_res_contour) that has been initially triangulated (initial_elements) the function returns a large scale uniform or adaptive mesh. The variable size is either a constant number that represents the target size of the elements in the case of uniform mesh generation or a sizing function in case of adaptive mesh generation. The variable surface_points represents the points of the high resolution contour upon points are projected during the refinement process. '''

def large_mesh_generation(low_res_contour,surface_points,surface,initial_elements,scale='uniform',size):
    contour_polygon=geom.Polygon(low_res_contour)
    surface_polygon=geom.Polygon(surface_points)

    call_counter=0
    edges_dict=dict()
    edge_list=[]
    elements_list=[]
    sub_contour_polygons=[]

    sub_contour_polygons.append(low_res_contour)
    elements_list.append(initial_elements)

    refine=True

    # Collect edges of initial elements
    edge_list=[]
    edges_dict_list=list()
    sub_contours=[]
    while refine:
        sub_contours=[]
        edge_list=list()
        edges_dict_list=list()
        for index,low_res_contour in enumerate(sub_contour_polygons):
            refined_edge_set=set()

        for element in elements_list[index]:
            edges_dict=dict()

            element_contour=low_res_contour[[element[0],element[1],element[2]]]
            element_contour_edges=[i for i in combinations(element,2)]
            edge_list.append(element_contour_edges)

            for edge in element_contour_edges:
                edge_coords=np.array([low_res_contour[edge[0]],low_res_contour[edge[1]]])
                edges_dict.update({str(edge):edge_coords})

                if scale=='uniform':
                    inserted_points=insert_point_edges_uniform(edge_coords, target_edge_length)
                elif scale=='adaptive':
                    inserted_points=insert_point_edges_adaptive(edge_coords, size_function)

                edges_dict.update({str(edge):inserted_points})
                if  len(edges_dict[str(edge)])>2:
                refined_edge_set.add(edge)
                edge_points=edges_dict[str(edge)]

            # if an edge is part of the high res contour, project inserted vertices to it
            for index,point in enumerate(edge_points):
                if index!=0 or index!=len(edge_points)-1:
                    is_near=contour_polygon.boundary.distance(convert_to_Point(point))<1e-8

                if is_on_surface(point,surface_points) and  is_on_surface(point, surface_points) and is_near:
                    point=convert_to_numpy(project_to_surface_point(point, surface_points,surface))
                    edges_dict[str(edge)][index]=point
        edges_dict_list.append(edges_dict)

    # Check if there are edges with inserted points
    if len(refined_edge_set)!=0:
        counter=-1
        for elements_index,initial_elements in enumerate(elements_list):
            for index,element in enumerate(initial_elements):
            subcontour_points=[]

            counter+=1
            edges=edge_list[counter]


            # Form sub_contours
            for  edge in edges:
                edge_points=edges_dict_list[counter][str(edge)]
```

```
                    for index,point in enumerate(edge_points):
                        for point in edge_points:
                            subcontour_points.append(point)

                            subcontour_points=np.array(subcontour_points)
                            subcontour_points=np.unique(subcontour_points, axis=0)
                            sub_contour_reshaped=subcontour_points.reshape(1,subcontour_points.shape[0],2)
                            sub_contour= sort_points(sub_contour_reshaped,subcontour_points.shape[0]).reshape(subcontour_points.shape[0],2)
                            if not is_counterclockwise(sub_contour):
                                sub_contour=sub_contour[::-1]
                            sub_contours.append(sub_contour)



            elements_lists=[]
            sub_contours_with_inner_points=[]
            inner_points_list=[]

            # Triangulate sub contours
            for subcontour in sub_contours:
                print(subcontour)

                # Triangulate sub-contour for the mean of its edge lengths
                target_edge_length=np.mean(get_edge_lengths(subcontour))

                # Call NN1 for prediction of number of inner points
                nb_of_points= get_nb_of_points(subcontour,target_edge_length)

                # Call NN2 for prediction of location of inner points
                inner_points= get_inner_vertices(subcontour,target_edge_length,nb_of_points)

                subcontour_with_inner_points=np.vstack([subcontour,inner_points])

                # Call NN3 for prediction of connectivity
                element_list=triangulate_NN(subcontour, inner_points)




                elements_lists.append(element_list)
                sub_contours_with_inner_points.append(subcontour_with_inner_points)

            # Smooth resulting mesh
            sub_contours_with_inner_points=smooth_recursive(sub_contours_with_inner_points, elements_lists)

            sub_contour_polygons=sub_contours_with_inner_points
            elements_list=elements_lists
```

```
        else:
            #If no edges with inserted points are found end further refinement
            refine=False
    return sub_contours, inner_points_list, sub_contours_with_inner_points, elements_lists
```

Alexis PAPAGIANNOPOULOS

Greek / French
B permit (CH)
Skype: alexis.papayannopoulos

Av. De la Gare 29,
1003 Lausanne –Switzerland
+41 77 452 52 18
alexis.papagiannopoulos@epfl.ch

Profile

Applied Mathematician specialized on computational geometry and machine learning. Passionate for programming languages and the certainty of Mathematics and Physical Sciences.

Strengths

- 7 years working in institutions for research
- 6 years experience in mathematical modelling and software development
- Fast learning, analytical thinking, and pragmatic
- Trilingual: Greek, English and French

Experience

- **Doctoral Research Associate**

  Former Laboratory of Hydraulic Machines Group (**LMH)** from École Polytechnique Fédéral de Lausanne **(EPFL),** Lausanne Switzerland
  2016-2020

  Laboratory of Fluid Mechanics and Instabilities (**LFMI)** from École Polytechnique Fédéral de Lausanne **(EPFL),** Lausanne Switzerland
  2020

- Main researcher in the investigative work of application of Neural Networks for mesh generation and improvement
- Development of algorithms using Python and C++ with the use of machine learning and mesh libraries.
- Guidance and supervision of research work for the development of mesh improvement application using my preset code.

- **Research Assistant**
  National Institute for Research in Computer Science and Control (INRIA) within the GALAAD Team, Nice, France
  2015

- Development of a plugin for AXEL (a CAD/CAE) using libraries such as Qt, Eigen and Axel for the extraction of mesh by parametrizing surfaces

- **Research Assistant**
  National Technical University of Athens, School of Naval Architecture and Naval engineering, Division of Ship Design and Maritime Transport, Athens, Greece
  2012 - 2015

- Main researcher around the applications of iso-geometric analysis on ship design
- CAD/CAE – oriented research. Continuation of the study of the isogeometric computational method focusing on the optimization of geometry for bodies (airfoils, ship hulls etc.) subjected to physical problems (flow problems).
- Use of C++ with mathematical and geometrical libraries, Software development using Matlab for the construction of a solver (finite elements, boundary elements), software development using Python for scripts which initialized an optimization loop and call for applications.

- **Mathematics and Physics Tutoring**
  Municipality of Zografou
  2013 - 2014

- Highschool level of Mathematics and Physics tutoring to the students of the Municipality

- **IT Internship**
  Hospitality Integrated Technologies A.E., Athens, Greece
  2008 - 2009

- Hands on work in applications using HTML and SQL

Education

M.Sc. Mathematical Modeling in Economy and Cutting-Edge Technologies
2009-2011
National University of Athens **(NTUA)**, School of Applied Mathematics and Physical Sciences, Athens, Greece
GPA: 7.9 / 10

Thesis: *Solving Partial Differential Equations using the method of Isogeometric Analysis*

Engineering Degree of Applied Mathematics
2003-2009
National University of Athens **(NTUA)**, Athens, Greece
GPA: 7..5 / 10

Specialization: *School of Applied Mathematics and Physical Sciences*

Studies Description: Main focus on theoretical math and the applications of mathematics in statistics; specially in the use of statistical package R, and information technology (Java applications, data structures, cryptography, algorithms and complexity)

Thesis: *Unbounded Operators and Application*

Description: *Work covering the field of theoretical math and functional analysis, particularly analyzing the Unbound operators as the differential Operator.*

Advisor: *Assoc. Prof. Sotiris Karanasios*

Languages

English
Fluent spoken and written

French
Upper Intermediate level spoken and written (B2)

Greek
Native Language

Technical Skills

Programing Skills: Python, C++, MATLAB, HTML, SQL
Libraries: Scipy, Pytorch, Tensorflow, Panda, CGal

Publications

- "How to teach neural networks to mesh: Application on 2-D simplicial contours" Submitted to Journal of Neural Networks (Elsevier). *Authors: A. Papagiannopoulos, P. Clausen, F. Avellan*
- "Local mesh improvement with the application of Neural Networks" (under preparation). *Authors: A. Papagiannopoulos, A. Flynn, P. Clausen, F. Avellan*
- "An isogeometric BEM solver for exterior potential flow problems around lifting bodies". 11th World Congress on Computational mechanics. *Authors: : C.G. Politis, A. Papagiannopoulos,K.A. Belibassakis, P.D. Kaklis, K.V Kostas, A.I.Ginnis, T.P.Gerostathis*

Personal interests

Concerts, Puzzles, Cinema, Programming