Author version

Published as:

Li, N., & Hwang, J. T. (2020). Automatic generation of global shell-element meshes for large-scale structural design optimization. In AIAA AVIATION 2020 FORUM (Paper No. AIAA 2020-3135). https://doi.org/10.2514/6.2020-3135

BibTeX for this reference:

https://lsdolab.github.io/lsdo-publications/publication/li2020automatic/

```
@inproceedings{li2020automatic,
  author = {Ning Li and John T. Hwang},
  title = {Automatic generation of global shell-element meshes for large-scale structural
    design optimization},
  booktitle = {AIAA AVIATION 2020 FORUM},
  year = {2020},
  doi = {10.2514/6.2020-3135},
  url = {https://doi.org/10.2514/6.2020-3135}
}
```

Automatic generation of global shell-element meshes for large-scale structural design optimization

Ning Li* and John T. Hwang†
University of California San Diego, La Jolla, CA, 92093

In aircraft design, unconventional configurations have gained significant recent interest due to increasing demands for greater efficiency as well as emerging applications such as urban air mobility. Since traditional empirical models are not applicable for unconventional aircraft, high-fidelity computational design methods like large-scale design optimization can be employed. However, building up models for such unconventional aircraft is time-consuming and requires a large amount of manual work during the iterative design process. Generating a mesh of a structure with a complicated geometry is one of the difficult steps. We present a fully automatic approach for constructing quadrilateral meshes for shell-element analysis of aircraft structures. The method is designed to produce highly isotropic meshes while guaranteeing coincidence of the nodes of the meshes of internal members and the structural skin.

I. Nomenclature

$\eth$	=	input graph from CAD software
ν	=	the set of vertices in the input triangle mesh
ϵ	=	the set of edges in the input triangle mesh
δ	=	the set of triangles in the input triangle mesh
$\boldsymbol{v}_i$	=	the coordinates of the vertex i
$\boldsymbol{N}(i)$	=	the neighborhood of vertex i
l	=	the ordered set of discretization points on the line segment
$\boldsymbol{l}_i$	=	the coordinates of discretization point i
p	=	the ordered set of projected points
$\boldsymbol{p}_i$	=	the coordinates of the projected point i
m	=	the middle point of discretization point i and $i+1$
$\boldsymbol{r}_0,\ \boldsymbol{r}_1,\ \boldsymbol{r}_2,\ \boldsymbol{r}_3$	=	four vertices of a rectangle
$x,\ y,\ z$	=	indices of ν
$f_s,\ f_m,\ f_sm$	=	the objective function for the splitting, splitting, and smoothing optimizations
$\boldsymbol{c}_s,\ \boldsymbol{b}_s$	=	the vectors of coefficients for the splitting optimization
$\boldsymbol{c}_m,\ \boldsymbol{b}_m$	=	the vectors of coefficients for the merging optimization
$\boldsymbol{x}_s,\ \boldsymbol{x}_m$	=	the decisions of the splitting and merging optimization
$\boldsymbol{lb}_s,\ \boldsymbol{up}_s$	=	the lower and upper bound for the design variables in the splitting optimization
$\boldsymbol{lb}_m,\ \boldsymbol{up}_m$	=	the lower and upper bound for the design variables in the merging optimization
$\boldsymbol{A}_s,\ \boldsymbol{A}_m$	=	the matrix of coefficients for the splitting and merging optimization
e_s	=	the energy to split a polygon
e_m	=	the energy of a quad
$c_1,\ c_2,\ c_3$	=	the coefficients determine e_s and e_m
l_i	=	the length of the ith edge in a polygon
l_avg	=	the average length of all the edges in a polygon
θ_i	=	the angle of the ith angle in a polygon
θ	=	the reference angle for a polygon
$\boldsymbol{e}_i$	=	the unit vector pointing along edge i of a quad

*Master's Candidate, Department of Mechanical and Aerospace Engineering.
†Assistant Professor, Department of Mechanical and Aerospace Engineering, AIAA Member.

$\boldsymbol{n}_q$	=	the normal vector of a quad
$\boldsymbol{V}_0,\ \boldsymbol{V}_1$	=	the initial position of the endpoints for all edges
$\boldsymbol{E}_0,\ \boldsymbol{E}_1$	=	the indices for endpoints of all edges
$\boldsymbol{D}$	=	displacement for all vertices
$\boldsymbol{m}$	=	matrix form of $\boldsymbol{D}$
$\boldsymbol{r}$	=	the maximum radius each vertex allowed to move
$\boldsymbol{n},\ \boldsymbol{A}$	=	is the matrix form of normal vectors for all vertices
$\boldsymbol{w}$	=	regularization parameter
$\boldsymbol{B}$	=	the partial derivative of the first term in the objective function with respect to $\boldsymbol{D}$
$\boldsymbol{D}^*,\ \boldsymbol{\lambda}^*$	=	the optimal displacements and associated Lagrange multiplier

II. Introduction

Large-scale design optimization (LSDO) is the solution of numerical optimization problems in the context of engineering design, where the optimization problem has a large number of design variables, at least in the hundreds. Such an approach is useful because optimization in high-dimensional design spaces yields results that are unintuitive and can provide insights into complex design problems. The high dimensionality can come from a problem that involves multiple coupled disciplines, as is frequently the case in aircraft design. It can also come from a problem in which the quantity being optimized is spatially distributed, such as the sizing of the different parts of a load-carrying structure.

In the field of aircraft design, there is currently a large amount of interest in unconventional aircraft configurations for: commercial aircraft to achieve radical improvements in fuel efficiency and environmental impact; supersonic aircraft to enable their re-introduction into the market; and electric vertical takeoff and landing (eVTOL) in the emerging area of urban air mobility (UAM). With unconventional configurations, traditional empirical and semi-empirical models are inadequate; therefore, high-fidelity computational design methods such as LSDO can play an important role. However, the challenge in applying high-fidelity tools to a conceptual design setting is satisfying the conflicting criteria that: the tool is versatile in that it can consider a wide range of designs with large design changes; there is a small turnaround time so that the engineer can quickly set up, execute, and visualize the computational results; and mesh quality is maintained so that the expected accuracy of the simulation is achieved.

The difficulty of satisfying all of these criteria simultaneously has thus far limited the use of high-fidelity tools early in the design process such as in conceptual design where unconventional configurations are considered. If these criteria can be satisfied, large-scale design optimization has the potential to significantly accelerate the investigation of unconventional configurations for which there is a lack of prior knowledge and empirical models.

The focus of this paper is on the structural analysis in large-scale design optimization (which can be multidisciplinary). Structures is an important discipline to consider even at the earlier stages of aircraft design because of the importance of reasonably accurate weight prediction. For conventional aircraft, there are many empirical weight models [1] broken down by part (e.g., wing, fuselage, horizontal tail, landing gear) which are functions of various parameters (e.g, area, aspect ratio, sweep, fuselage length, gross weight). These empirical models are very useful because they are simple and easy to use, but they provide enough accuracy to obtain reasonable early estimates of empty weight and mass distribution. For unconventional configurations, we desire structural analysis methods and tools that can provide a similar ability to estimate empty weight and mass distribution while satisfying the aforementioned criteria: versatility with respect to large design changes; small turnaround time; and reasonable mesh quality. High-fidelity structural simulations of airframes are performed using shell-element finite-element analysis.

The generation of a shell-element mesh of an aircraft can be challenging since the internal members of the aircraft affect the mesh of the aircraft's outer skin because the nodes of the meshes of internal members and structural skin must coincide. Jakob et al. [2] introduce a set of interactive brush tools that can be used to control the alignment of the edges in the final mesh, their exact position on the surface, e.g. the location of an imprinted member of an aircraft can not be easily added with brush tools exactly. Hwang et al. [3] develop the GeoMACH tool suite to generate geometry and meshes while allowing the users to determine the layout of the internal structures and keeping the coincidence of intersection nodes between the members and skin. However, GeoMACH has several limitations: it is very time-consuming; its algorithm requires the geometry to be developed within GeoMACH and cannot work with externally created geometries such as CAD files; and it requires a considerable amount of manual work since it subdivides the entire geometry surface into components and generates meshes on these components respectively to combine them into the mesh of the surface [4]. Thus, a simple approach is desired that can generate a high-quality

mesh of an airframe without losing automation and robustness. OpenVSP (Open Vehicle Sketch Pad), is NASA's parametrically driven, open-source aircraft geometry design tool, which can be used to generate triangle meshes and quadrilateral wireframes [5]. However, OpenVSP can only generate triangular meshes of thin-walled aircraft structures, and its capability for generating quadrilateral meshes is limited to structured wireframes. Here, we desire a method that can generate fully unstructured quadrilateral meshes of the entire airframe including the skin and internal members of wings, fuselages, and other aircraft components.

In this paper, we present a novel approach that satisfies all previous requirements using an algorithm whose key steps are a series of simple and robust solutions of linear and quadratic programming problems. Compared to nonlinear programming, linear and quadratic programming are always more robust, simple, and efficient. Our fully automatic mesh algorithm starts with the model of a commercial CAD software, e.g., SolidWorks. For each member, it first defines four intersection curves that become boundaries of the internal member by projecting four line segments out of the aircraft surface to the aircraft model. Then, we apply transfinite interpolation (TFI) to create a continuous parametric surface based on the four boundary curves and generate the initial mesh nodes in the parametric domain [6]. The mesh nodes are later used to generate an initial mesh using Delaunay triangulation, which always has a unique solution in the 2D case. After that, the intersection points between the projected curve and the original triangulation is found and used to reconstruct the triangulation. At last, several mesh quality improvement steps are performed on the meshes of the imprinted members and aircraft skin separately. The mesh quality improvement methods involve splitting, merging, and smoothing optimizations, which are formulated as linear or quadratic programming problems to make topological and geometric adjustments.

The paper proceeds as follows. In Sec. III, we survey existing related work in mesh generation of aircraft structures, and in Sec. IV, we demonstrate the details of our algorithm. In Sec. V, we present the results of applying this algorithm to a model from Vehicle Sketch Pad (OpenVSP). In Sec. VI, we summarize the accomplishments of this paper and future plans.

III. Related work

A wealth of methods and algorithms have been developed to generate shell-element meshes. For maximizing mesh quality, much of the prior work focuses on minimizing energy to capture smoothness. Some previous studies are able to generate high-quality quadrilateral meshes by minimizing the smoothness energy [2, 7, 8]. Knoppel et al. [9] further generate an algorithm for computing the stripe patterns on a surface which can be applied to create a mesh on the surface. By extending the idea of smooth orientation and position field [2], Huang et al. [10] remove most of the singularities in the output mesh by formulating a minimum-cost network flow problem However, they are all complicated and some of them involve nonlinear programming which doesn't guarantee convergence or global minimum of the objective function. The mesh generated by the algorithm proposed in this paper may not always be better, but it is more efficient and robust.

In addition to the prior work that focuses on minimizing the smoothness energy, other methods have been developed to generate the mesh or smooth the existing mesh. Parthasarathy et al. [11] develop a constrained optimization method to generate a smooth mesh. Canann et al. [12] use a local optimization approach to improve the topology of unstructured quadrilateral finite element meshes. Some prior work has also been done to use the parameterization of surface patches to generate quadrilateral meshes by initializing the mesh in the parametric domain, then do the smoothing and project the mesh back to physical domain. [13–15]. After the projection, some post-processing steps may be implemented to improve the mesh quality. One drawback of using parameterization to generate meshes is that if the optimization and smoothing are done in the parametric domain, the projected meshes in physical domain may be distorted. Our optimizations are done in the physical domain, thus the mesh quality is preserved. Verma et al. [16] provide a method to produce an isotropic fully-quad mesh using Suneeta Ramaswamy's tree matching algorithm and Guy Bunin's one-defect remeshing. Several studies focus on generating mesh from geometries with gaps or overlaps, or fixing inter-domain boundaries [17–19].

Recent papers focus more on the adoption of existing algorithms and incorporation of the existing algorithms and techniques from the other fields. Some papers adapt the advancing front techniques to generate quadrilateral meshes [20–25]. The advancing front techniques are robust and simple, but it is usually more time-consuming compared to other global mesh generation methods. Rouxel et al. [26] define a discrete Riemannian Voronoi diagram for which its dual is an embedded triangulation. Engwirda et al. [27] offer a robust Frontal-Delaunay approach in which new vertices are constrained by element size and the shape of the model.

Otoguro et al. [28] present a method based on multiblock-structured mesh generation and projection of the structured

mesh to a NURBS mesh, and recovery of the original model surfaces.

Liu et al. [29] combine an octree-based polyhedral mesh generation with the scaled boundary finite element method to perform stress analysis of standard tessellation language (STL) models. The octree structure is a commonly used data structure for 3D models, which can help improve the efficiency of searching desired elements. Some researchers introduce more metrics to the existing algorithms to improve the mesh quality [30, 31]. Soni et al. [32] propose an efficient algorithm by applying centroidal voronoi tessellation on the voxelized Surface. Altenhofen et al. [33] model 3D objects using a volumetric subdivision representation to encode the volumetric information in the design mesh making the mesh generation process much faster. CDT is an extension of DT by constraining the desired connections of vertices.

What is lacking in the prior work is an efficient way to add internal members to a model, which is one of the motivations of the paper. The algorithm that we propose as a solution to this is based partly on the algorithm in the aforementioned GeoMACH library [3]. This algorithm divides the aircraft skin into faces and generates a smooth mesh on each face based on a six-step algorithm. However, the generated mesh is considered smooth locally in the parametric domain of each face. The algorithm proposed here can generate smoother meshes since all the optimizations including smoothing are done globally in the physical domain. Another weakness of GeoMACH is that it requires the user to generate the geometry within GeoMACH which is time-consuming. Also, the given CAD geometries cannot be reproduced exactly in GeoMACH. The current algorithm works with input geometries specified by CAD files.

IV. Methodology

The algorithm begins with a triangulation of the geometry, which would in most cases be generated by commercial CAD software. The triangle mesh can be represented as $\eth = (\nu, \epsilon, \delta)$, where ν is the set of all vertices associated with a position $\boldsymbol{v}_i \in \mathbb{R}^3$. In addition, ϵ is the set of edges of the input mesh, $(i, j) \in \epsilon$ if $\boldsymbol{v}_j$ is the neighboring vertex of $\boldsymbol{v}_i$. Similarly, $(i, j, k) \in \delta$ if $\boldsymbol{v}_i$, $\boldsymbol{v}_j$ and $\boldsymbol{v}_k$ forms a triangle on the surface mesh. The neighborhood of vertex i is defined by $\boldsymbol{N}(i) = \{j \in \nu | (i, j) \in \epsilon\}$.

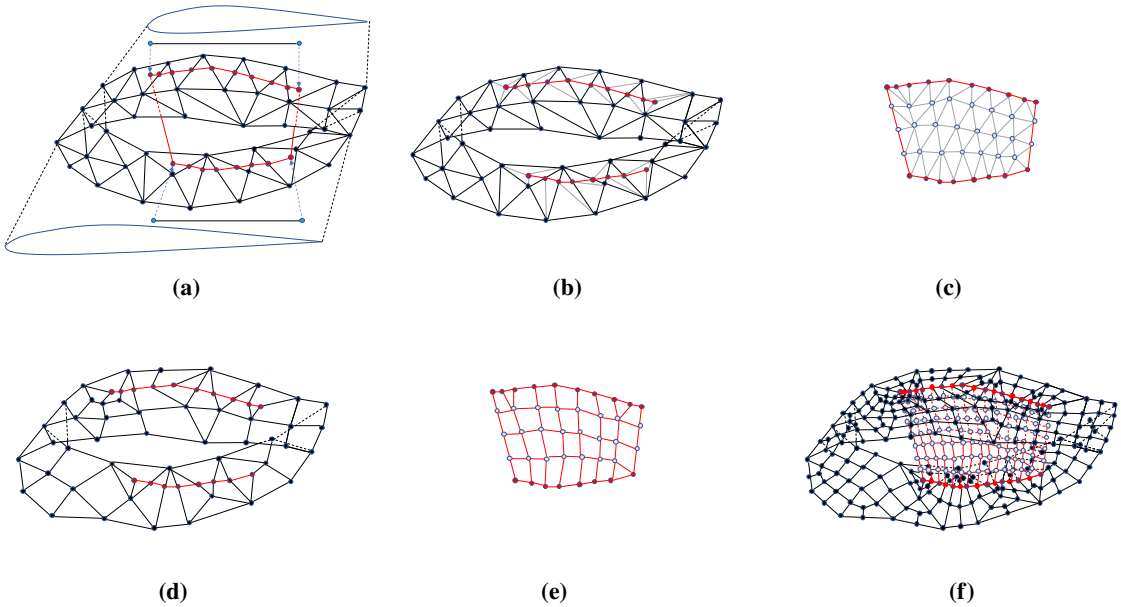


Fig. 1 Overall flow of the algorithm: (a) projection of points (b) construction of triangulation (c) creation of a rib (d) implementation of the mesh quality improvement methods on the wing skin (e) implementation of the mesh quality improvement methods on the rib (f) generation of fully quad mesh

We present the algorithm by first providing an overview in Sec. IV.A. Next, we describe in more detail one of the steps, which is the addition of points and reconstruction of the triangulation in Sec. IV.B. Finally, we describe in Sec. IV.C how the mesh quality is improved via merging, splitting, and smoothing optimizations .

A. Algorithm overview

To generate the mesh of the skin, we first compute the intersection curve between the internal members and the outer skin of the model by projecting two or more line segments onto the triangulation as shown in Fig. 1a. Then, we can reconstruct the triangulation to involve all the projected points presented in Fig. 1b. Internal members are defined as four-sided surfaces with each side created from the projection of a discretized line segment. By interpolating the four sides with B-splines curves and applying TFI, we can create a parametric surface of the member [6]. The initial meshes of the internal members are created by generating evenly distributed points and utilizing the DT method in the parametric domain (Fig. 1c). Afterwards, the mesh quality improvement methods are implemented on the meshes of the skin and the internal members separately (Fig. 1d and 1e). Finally, the quad-dominant mesh is transformed into a fully quad mesh with one step of the Catmull–Clark subdivision (Fig. 1f). Optionally, we can still use the mesh quality improvement methods on the fully quad mesh. If more than one members are created, we interactively check if any two of these members intersect, if so, each of these two intersected members will be split into two sub-members by the intersection line. Then, the intersection lines are constrained for each member to make sure that after optimization, these intersection lines still match up.

The method can be split into two parts, the addition of points and reconstruction of the triangulation, and mesh quality improvement methods. In the first part, a detailed explanation of two projection algorithms and the connection of added points are presented. In the second part, we systematically introduce the formulation of two topological optimization and one geometric optimization problems to improve the quality of the shell element mesh.

B. Addition of points and reconstruction of the triangulation

Given the line segments that decide the layout of the internal members, we discretize the line segment into several pieces evenly or maintain the original line segment depending on the targeted mesh resolution. Then for each discretization point $\boldsymbol{l}_i \in \mathbb{R}^3$ with $i \in l$, l is the set of all the points on the line segment, we find the corresponding projected point $\boldsymbol{p}_i \in \mathbb{R}^3$ with i being a member of the set storing all the projected points p. All elements in l and p are ordered so that every two adjacent members in each set are neighboring points in the line segment or projected curve. Considering that the imported model may be high-fidelity and have complicated geometry with a significant number of vertices, we use the octree data structure to store all the triangles in the triangulation based on their positions to increase the speed of search as shown in Fig. 2. Two algorithms can be employed to project the discretization points onto the surface of the model. If the projection directions are given, we can define a ray starting from the point on the line pointing along the projection direction and find the intersection point with the model that gives the smallest distance between the intersection and projection point. Starting from the root node in the octree [*], we can first determine the probability of the intersection of the ray and a node, which is a cubic box in the octree. If the intersection is possible, we loop through all the subnodes and check the occurrence of the intersection. By doing this operation recursively, we end up with a leaf node. Looping through all the triangles in the leaf node and finding the triangles that intersect with the ray, the distances between the triangles and projection point and projected points can be found. Among all the projected points, the one that yields the smallest distance is the point we want. If the projection direction is not given, we can then find the point on the triangulation that gives us the smallest distance to the projection point. The algorithm is not too much different from the previous one with projection directions. The first step is that from the root node, we iterate over all the 8 subnodes and find the one that has the smallest distance to the projection point. If it is a leaf node, we loop through all the triangles in this node and find the projected point and corresponding smallest distance. If it is not a leaf node, we do the first step until we end up with a leaf node. At each level, we need to check if the found smallest distance is smaller than the distance from the projection point to the other nodes. If the found distance is smaller compared to that between the projection point and the other nodes, go to the upper level and check again. If not, we need to find the smallest distance between the triangles in the other node and compare it to the found smallest distance in case that the projected point is in the node that doesn not yield the smallest distance between it and the projection point among all its counterparts.

[*] https://github.com/mhogg/pyoctree

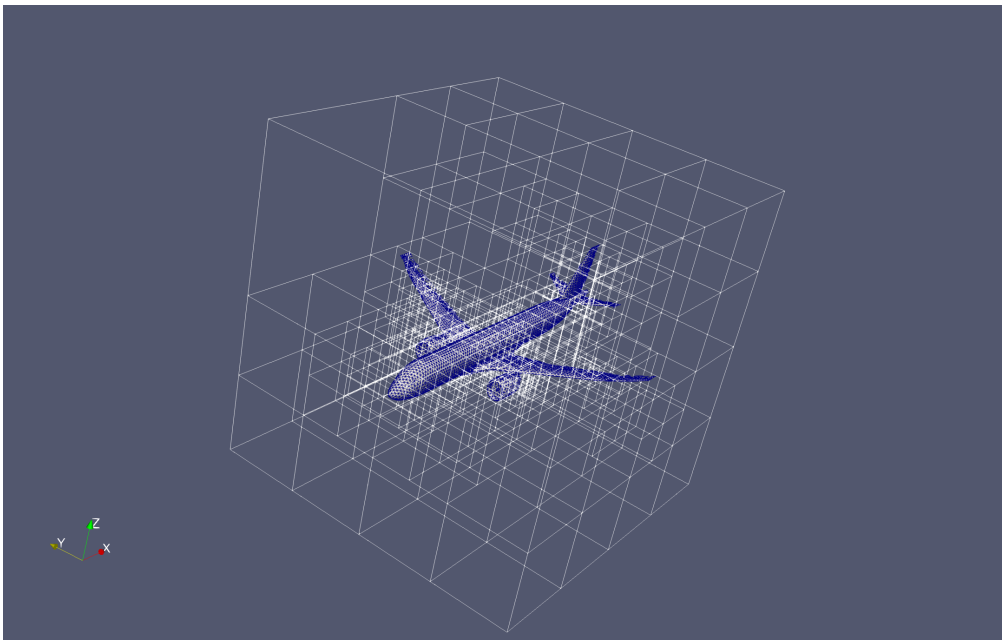


Fig. 2 Details of the projection method: The generated octree structure for an aircraft contains the aircraft in it. The boxes are the nodes of the octree. Depending on the density of triangles, the octree nodes can be coarser or finer.

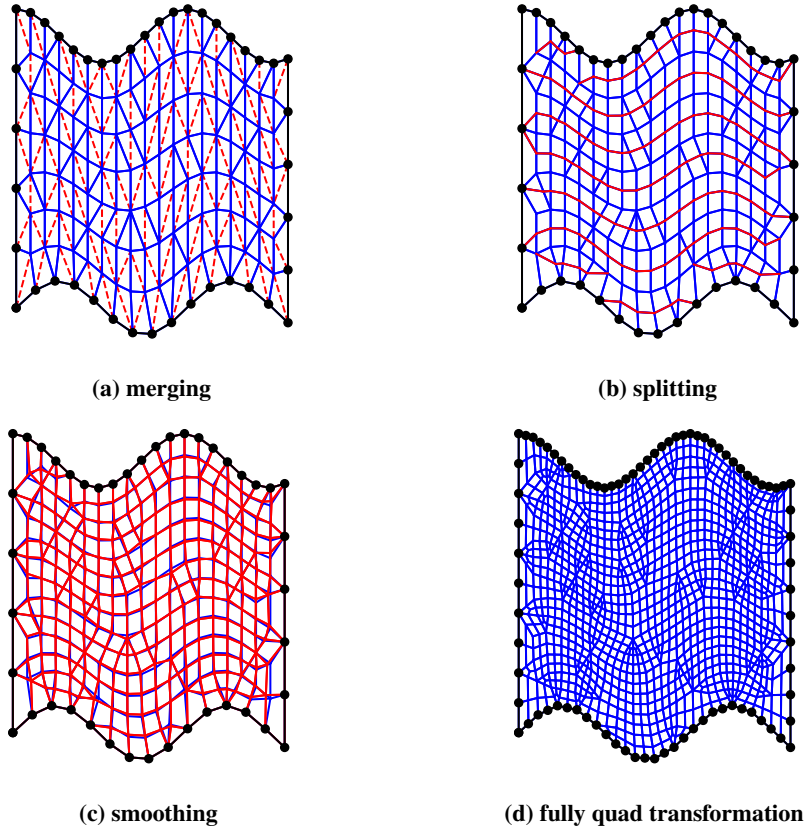
Once all the projected points of the discretized points on the line segment are found, we can connect them sequentially and reconstruct the triangulation. For each pair of neighboring projected points $\boldsymbol{p}_i$ and $\boldsymbol{p}_{i+1}$, we first calculate the middle point $\boldsymbol{m}$ of their corresponding projection points $\boldsymbol{l}_j$ and $\boldsymbol{l}_{j+1}$. Based on $\boldsymbol{p}_i$, $\boldsymbol{p}_{i+1}$, and m_i, a rectangle can be formed by creating an edge going through $\boldsymbol{m}$ parallel to the edge connecting $\boldsymbol{p}_i$ and $\boldsymbol{p}_{i+1}$ with m_i being the middle point of the edge it lies on. Then, the intersection points between all the edges and the created rectangle can be found easily and inserted between $\boldsymbol{p}_i$ and $\boldsymbol{p}_{i+1}$ with their distances to $\boldsymbol{p}_i$ in an increasing order. After inserting all the intersection points between every pair of neighboring projected points, p is updated. To construct the new triangulation, we iterate over all the members in p. If it locates on the edge of a triangle, we connect it and the opposite vertex in the triangle, and thus two new triangles are added to δ while the split triangle is deleted from δ. If it locates on the interior of a triangle, we connect it to all the three vertices of the triangle, therefore in this case, three triangles are added to δ and the split from δ. A detailed algorithm of how to reconstruct the triangulation is presented in Alg. 1.

Algorithm 1: reconstruction of the triangulation

Data: p, l, v, ϵ, δ
Result: p, δ
for *i in p except the last element, j in l except the last element* **do**
  $\boldsymbol{r}_0, \boldsymbol{r}_1 \leftarrow \boldsymbol{p}_i, \boldsymbol{p}_{i+1}$; // $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$ are the first two vertices of the rectangle
  $\boldsymbol{m} \leftarrow \frac{1}{2}\boldsymbol{l}_j + \frac{1}{2}\boldsymbol{l}_{j+1}$;
  $\boldsymbol{r}_2 \leftarrow \boldsymbol{m} + \frac{1}{2}(\boldsymbol{p}_{i+1} - \boldsymbol{p}_i)$; // the third vertex
  $\boldsymbol{r}_3 \leftarrow \boldsymbol{m} - \frac{1}{2}(\boldsymbol{p}_{i+1} - \boldsymbol{p}_i)$; // the fourth vertex
  for (x, y) *in* ϵ **do**
    if *line segment connecting* $\boldsymbol{v}_x$ *and* $\boldsymbol{v}_y$ *intersects the rectangle* **then**
      $\boldsymbol{coord} \leftarrow$ the intersection point between the line segment connecting $\boldsymbol{v}_x$ and $\boldsymbol{v}_y$ and the rectangle;
      $\boldsymbol{dist} \leftarrow \sqrt{(\boldsymbol{p}_i - \boldsymbol{coord})^2}$;
      save ($\boldsymbol{coord}$, $\boldsymbol{dist}$) in $\boldsymbol{int}$;
  end
  sort $\boldsymbol{int}$ with $\boldsymbol{dist}$ in the increasing order;
  insert $\boldsymbol{coord}$ from $\boldsymbol{int}$ between i and $i+1$ in p; // update $\boldsymbol{p}$ with intersection points
end
add p to v;
for *i in p* **do**
  for *(x, y, z) in* δ **do**
    if $\boldsymbol{v}_i$ *locates on the edge of triangle (x, y, z)* **then**
      delete (x, y, z) from δ;
      if $\boldsymbol{v}_i$ *lies between* $\boldsymbol{v}_x$, $\boldsymbol{v}_y$ **then** add (y, z, i) and (i, z, x) to δ;
      if $\boldsymbol{v}_i$ *lies between* $\boldsymbol{v}_y$, $\boldsymbol{v}_z$ **then** add (z, x, i) and (i, x, y) to δ;
      if $\boldsymbol{v}_i$ *lies between* $\boldsymbol{v}_z$, $\boldsymbol{v}_x$ **then** add (x, y, i) and (i, y, z) to δ;
    if $\boldsymbol{v}_i$ *locates on the interior of triangle (x, y, z* **then**
      delete (x, y, z) from δ;
      add (x, y, i), (y, z, i), and (z, x, i) to δ
  end
end

C. Mesh quality improvement

In this section, three optimization problems are formulated to improve the mesh quality, in which the splitting and merging optimization are the topological optimization, and the smoothing optimization is the geometric optimization as shown in Fig. 3. By applying the splitting and merging optimization to the mesh, the number of vertices in the mesh may increase, and the total number of polygons in the mesh also changes, which is caused by splitting a quad or a triangle and merging two adjacent triangles. On the other hand, the smoothing optimization step does not change the number of vertices or connectivities of polygons but moves the positions of the vertices. All the optimization methods aim to decrease the aspect ratio of the mesh or make the polygons more regular.



(a) merging **(b) splitting**

(c) smoothing **(d) fully quad transformation**

Fig. 3 Important steps in the mesh improvement methods: In all the subfigures, the black lines and points represents the fixed vertices and edges. In merging optimization (Fig. 3a), the red dashed lines are merged edges and the blue edges are the remaining edges. In splitting optimization (Fig. 3b), the blue edges are the original edges and red lines are the lines that splitting the polygons. In the smoothing optimization (Fig. 3c), the blue edges are the input edges, red edges are the smoothed edges. Some of the blue edges are covered by the red edges which means the edge does not change too much before or after the smoothing optimization. In the fully quad transformation (Fig. 3d), the smoothed quad-dominant mesh is further split to form a fully quad mesh.

1. Splitting optimization

The meshes extracted using commercial CAD software are usually structured. For structures with complex geometry, like aircraft, engineers often split the entire model into several components, for instance, fuselage, wings, and tail, and build these components individually. This is efficient when it comes to building up the model. However, the extracted shell element meshes may have bad quality because the CAD software tends to generate meshes for each component and merge them, which sometimes results in triangles with large aspect ratios at the intersection regions of different components. Similarly, after the retriangulation in our method to add intersection curves between the model outer surface and internal members on the outer surface, we kind of brutally connect the projected points to the vertices of the triangles in which these projected points locate, which also causes some triangles with bad quality. To deal with these irregular triangles and also expand the idea to quad element, the splitting optimization is derived to allow triangles and quads to be split into a triangle and a quad or two triangles or two quads, thus makes the polygons in the mesh become more regular. We form the splitting optimization problem as a linear programming problem for robustness and efficiency. The detailed expression is shown below:

$$\begin{aligned} \textbf{minimize} \quad & f_s = \boldsymbol{c}_s^{\mathsf{T}} \boldsymbol{x}_s \\ \textbf{with respect to} \quad & \boldsymbol{lb}_s < \boldsymbol{x}_s < \boldsymbol{ub}_s \\ \textbf{subject to} \quad & \boldsymbol{A}_s \boldsymbol{x}_s = \boldsymbol{b}_s, \end{aligned} \tag{1}$$

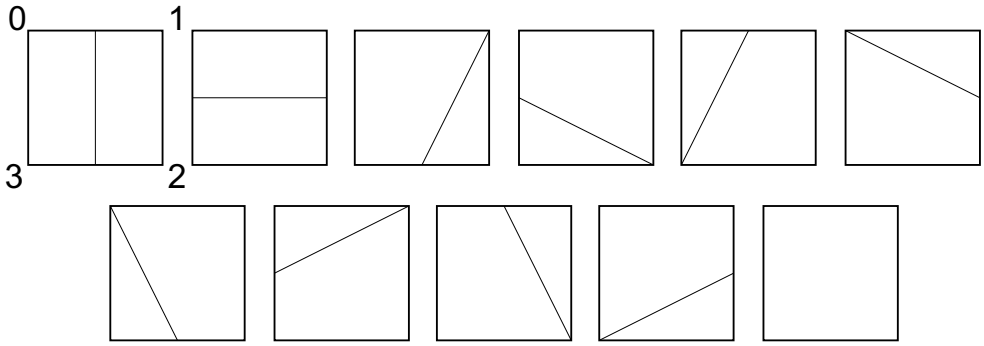


Fig. 4 Details of the quad splitting: There are eleven options to split a quad, the last one being the original configuration, not splitting at all. The number surrounding the top left quad (option 0) is the local indexing of the quad. From the top left to the bottom right, the options are indexed from 0 to 10.

where the design variables $\boldsymbol{x}_s$ has the size of eleven times the number of polygons in the mesh, eleven being the number of splitting options, lower and upper bounds $\boldsymbol{lb}_s$ and $\boldsymbol{ub}_s$ are usually 0 and 1, however in some cases the $\boldsymbol{ub}_s$ may be constrained to 0. The equality constraints are built up with 2 criteria, agreement on splitting the shared edge and only one option chosen for one polygon. More details are presented in the following paragraphs.

As shown in Fig. 4, the eleven options to split a quad can be divided into three types, splitting by a line connecting middle points of two opposite edges, splitting by a line connecting a vertex and the middle point of one opposite edge, and not splitting. As we want to make our polygons as regular as possible, we define energy describing the regularity of the mesh based on the aspect ratio of each polygon and the difference between the angle in each triangle or quad and 60 or 90 degrees. The expression is shown as:

$$e_s = c_1 \sum_{i=0}^{n} \frac{l_i}{l_{avg}} + c_2 \sum_{i=0}^{n} \frac{\theta_i}{\theta} \tag{2}$$

where e is the energy, n is the number of edges of a polygon, l_i is the length of each edge, l_avg is the average length of all the edges in this polygon, θ_i is the angle of each corner, c_1 and c_2 are the coefficients for energy regarding edge lengths and angles, and θ is the reference angle, 60 degrees for triangles and 90 degrees for quads. The weight, referred to as how much we prefer an option, can be calculated as the total energy after the splitting operation. For instance, if we split a quad into two new quads, the weight is calculated as the sum of the energy of two new quads. After calculating the weights of all the options, we group them up as a vector and assemble it to $\boldsymbol{c}_s$ in Eqn. 1.

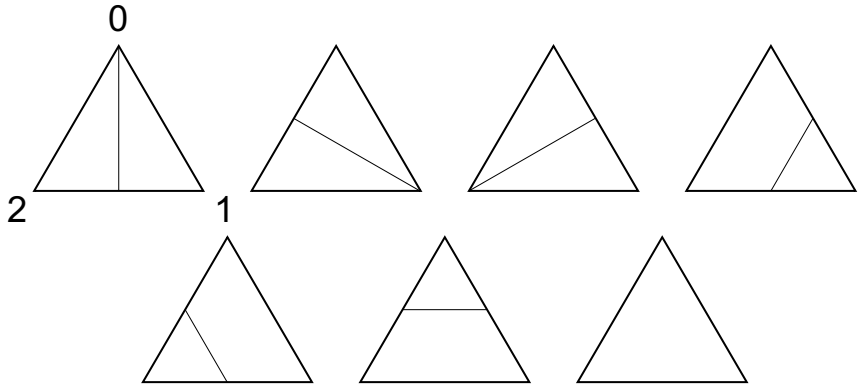


Fig. 5 Details of the triangle splitting: There are seven options to split a triangle, the last one being the original configuration, not splitting at all. The number surrounding the upper left triangle (option 0) is the local indexing of the triangle. From the top left to the bottom right, the options are indexed from 0 to 6.

There are fewer options for splitting a triangle, but we still assign eleven options to it since we want to make the

number of options the same for a triangle and a quad to make the assembly of $\boldsymbol{c}_s$ easier. Thus we need to change the upper bounds of the last four triangle splitting options to zero for validity. The computation of the weight of each option of a triangle is the same as that of a quad. The design variable $\boldsymbol{x}_s$ is the decision of each option for all polygons, thus its length is eleven times the number of polygons.

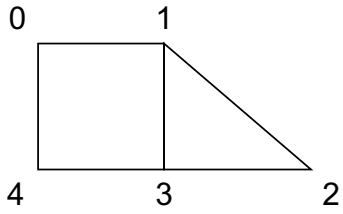


Fig. 6 An example showing the formulation of the equality constraint: The numbers are the indices for each vertex in the mesh. The quad can be presented as $(0, 1, 3, 4)$ **and the triangle can be expressed as** $(1, 2, 3)$**. They have one shared edge.**

Since every option is in contrast to all other options and the polygon is either split or not split. One equality constraint can be formed as the sum of all options decisions must be one. To this point, we only consider the situation of each polygon and overlook the adjacency of them, another equality constraint for establishing agreement on whether to split a shared edges of adjacent polygons must be constructed. Take the simple mesh in Fig. 6 as an example, the quad $(0, 1, 3, 4)$ and triangle $(1, 2, 3)$ have one shared edge $(1, 3)$. Since they have the shared edge, they must agree on whether to split the shared edge. For the triangle, the local indices of shared edge are $(2, 0)$ from Fig. 5, options $(2, 3, 5)$ split the shared edge and options $(0, 1, 4, 6)$ do not split it, while for the quad, the local indices of shared edge are $(1, 2)$, options $(0, 2, 6)$ tend to split the shared edge, options $(1, 3, 4, 5, 7, 8, 9, 10)$ do not split it. Therefore, as the splitting decisions agree between the quad and triangle, we can formulate a linear equality constraint:

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & -1 & -1 & 0 & -1 & -1 & -1 & -1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{x}_q \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \tag{3}$$

where $\boldsymbol{x}_t$ is the decisions for the triangle and $\boldsymbol{x}_q$ is the decisions for the quad, the first row of the first matrix means the triangle and the quad agree to split the shared edge while the second row represents that they agree on keeping the shared edge.

Sometimes we also have a list of edges that we want to fix during the splitting optimization stemming from the intersection curve of the internal members and model skin or vertices at the high-curvature region. To fix these edges, we find the corresponding edges in the triangulation and set the upper bound of every option that tends to split the fixed edge to zero. Thus, the fixed edge cannot be split after the optimization.

2. Merging optimization

Merging optimization is designed to merge two adjacent triangles to a high-quality quad to create a quad-dominant mesh. The best part of it is that if the given mesh is a structured mesh, although each triangle may be irregular, the merging optimization can help to transform it into a quad-dominant mesh that each quad in it is close to a square. The merging optimization can also help to decrease the number of edges, polygons which helps to decrease the computational cost of the following optimization steps. To control the resolution of the final mesh, merging optimization is critical. Just like the splitting optimization, the merging optimization problem can also be formulated as a linear optimization problem. The detailed expression is presented as:

$$\begin{aligned} \textbf{minimize} \quad & f_m = \boldsymbol{c}_m^{\mathsf{T}} \boldsymbol{x}_m \\ \textbf{with respect to} \quad & \boldsymbol{lb}_m < \boldsymbol{x}_m < \boldsymbol{ub}_m \\ \textbf{subject to} \quad & \boldsymbol{A}_m \boldsymbol{x}_m \leq \boldsymbol{b}_m, \end{aligned} \tag{4}$$

where the design variable $\boldsymbol{x}_m$ is based on each edge and its length is the number of edges, lower $\boldsymbol{lb}_m$ and upper bounds $\boldsymbol{ub}_m$ are set to be 0 and 1, 1 being merging the edge while 0 being not merging it. The inequality constraint stems

from the criteria that at most one edge from per triangle can be merged since if we allow multiple edges to be merged in a triangle, a polygon with more than four edges may be created. Since our algorithm can only handle triangle, quad-dominant, and fully quad mesh, we do not want it to happen.

Just like the splitting optimization, energy is defined to describe how good the merged quad can be. It has three terms, two are exactly the same as the energy defined for the splitting algorithm, the third one comes from the penalty of the dihedral angle of the resultant quad, which is shown as:

$$e_m = c_1 \sum_{i=0}^{n} \frac{l_i}{l_{avg}} + c_2 \sum_{i=0}^{n} \frac{\theta_i}{\theta} + c_3 \sum_{i=0}^{n} \boldsymbol{e}_i^{\intercal} \boldsymbol{n}_q, \tag{5}$$

where c_3 is the coefficient for the third energy regarding the dihedral angle, $\boldsymbol{e}_i$ is the unit vector pointing along each edge of the quad, $\boldsymbol{n}_q$ is the normal vector of the quad calculated as the cross product of the unit vectors in the two diagonals directions. The weight is computed by subtracting the energy before merging from the energy after the merging. The lower the energy after merging and the higher the energy before merging, the shared edge is more encouraged to be merged. Assembling the weight, $\boldsymbol{c}_m$ can be constructed.

The upper bound for an edge shared by two triangles is set to be 1, however, if the edge is found to be shared by two quads or a quad and a triangle, there is no way that we can merge them, thus the upper bound is set to be zero in this two cases. Also, if an edge is fixed, the upper bound for this decision of this edge is zero. To build up the matrix $\boldsymbol{A}_m$ and $\boldsymbol{b}_m$ in the inequality constraint, we first loop through all the edges to find the indices for edges in each triangle. Then we end up with a matrix that has a size of the number of triangles by three. By assigning corresponding elements in $\boldsymbol{A}_m$ as one with respect to the generated matrix and setting all the elements in $\boldsymbol{b}_m$, the inequality constraint is constructed.

3. Smoothing optimization

As described before, the splitting and merging optimization tends to optimize the mesh quality locally. The splitting optimization, although constrained by the criteria that the adjacent polygons must agree on the operation on the shared, still tends to focus on the quality of each element. Hence, the resultant mesh may have different element sizes in different regions, and the global aspect ratio may be even more significant than that before the splitting optimization. Similarly, in the merging optimization if at some regions, the mesh is formed with large triangles and at the other regions the mesh consists of small quads, then the global aspect ratio may remain large after the merging optimization. To deal with the problems derived from the splitting and merging optimization, the smoothing optimization step aims to change the location of each vertex in the mesh to decrease the aspect ratio, thus improve the overall quality. We allow all the vertices to move in its tangential plane within a small range, then minimize the sum of lengths of all edges with respect to the displacement vector Finally, we project all the vertices back to the model to maintain the geometry. The smoothing optimization method can be formulated as a quadratically constrained quadratic programming problem:

$$\begin{aligned} \textbf{minimize} \quad & f_{sm} = \frac{1}{2}[(\boldsymbol{V}_0^{\intercal} + \boldsymbol{D}^{\intercal}[\boldsymbol{E}_0]) - (\boldsymbol{V}_1^{\intercal} + \boldsymbol{D}^{\intercal}[\boldsymbol{E}_1])] \cdot [(\boldsymbol{V}_0 + \boldsymbol{D}[\boldsymbol{E}_0]) - (\boldsymbol{V}_1 + \boldsymbol{D}[\boldsymbol{E}_1])] \\ \textbf{with respect to} \quad & \boldsymbol{D} \\ \textbf{subject to} \quad & \boldsymbol{m}\boldsymbol{D} \leq \boldsymbol{r} \\ & \boldsymbol{n}\boldsymbol{D} = \boldsymbol{0}, \end{aligned} \tag{6}$$

where $\boldsymbol{D}$ is the displacement for all vertices, $\boldsymbol{V}_0$ and $\boldsymbol{V}_1$ are the initial position of the endpoints for all edges, $\boldsymbol{E}_0$ and $\boldsymbol{E}_1$ are the indices for endpoints of all edges, $\boldsymbol{V}_0 + \boldsymbol{D}[\boldsymbol{E}_0]$ and $\boldsymbol{V}_1 + \boldsymbol{D}[\boldsymbol{E}_1]$ are the updated position of the endpoints of all edges. The design variable $[(\boldsymbol{V}_0 + \boldsymbol{D}[\boldsymbol{E}_0]) - (\boldsymbol{V}_1 + \boldsymbol{D}[\boldsymbol{E}_1])]$ is calculated by subtracting the updated ending points from the updated starting points representing the vector along the edge and the objective function measures the lengths of edges. In the inequality constraint, $\boldsymbol{m}$ is matrix form of $\boldsymbol{D}$ by iteratively assigning three coordinates of each vertex to the corresponding positions in each row. Multiplying $\boldsymbol{m}$ and $\boldsymbol{D}$, we can measure the squared distance that each vertex moves smaller than $\boldsymbol{r}$, the maximum radius each vertex allowed to move, which is computed by the distance between the closest neighboring vertex and itself times a coefficient. In the equality constraint, $\boldsymbol{n}$ is the normal matrix created by assigning the normal vector of each vertex to the associated positions of each row. The equality constraint helps to keep each vertex move in its own tangential plane by setting the displacement vector vertical to the normal vector, which is calculated as the area-weighted normal of adjacent triangles for each vertex.

If we take the inequality constraint as a regularization term, the QP problem can be simplified to:

$$\begin{aligned} \textbf{minimize} \quad & f_{sm} = \frac{1}{2}[(\boldsymbol{V}_0^\intercal + \boldsymbol{D}^\intercal[\boldsymbol{E}_0]) - (\boldsymbol{V}_1^\intercal + \boldsymbol{D}^\intercal[\boldsymbol{E}_1])] \cdot [(\boldsymbol{V}_0 + \boldsymbol{D}[\boldsymbol{E}_0]) - (\boldsymbol{V}_1 + \boldsymbol{D}[\boldsymbol{E}_1])] + \frac{1}{2}\boldsymbol{D}^\intercal \boldsymbol{w} \boldsymbol{D} \\ \textbf{with respect to} \quad & \boldsymbol{D} \\ \textbf{subject to} \quad & \boldsymbol{n}\boldsymbol{D} = \boldsymbol{0}, \end{aligned} \tag{7}$$

where $\boldsymbol{w}$ is the regularization parameter decided by the local curvature for each vertex, and if some vertices are fixed or the curvature is found to be very high, we set the weight to be a extremely large number to prevent any movement. Eqn. 7 can be solved applying the KKT matrix:

$$\begin{bmatrix} \boldsymbol{B} & \boldsymbol{A}^\intercal \\ \boldsymbol{A} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{D}^* \\ \boldsymbol{\lambda}^* \end{bmatrix} = \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{c} \end{bmatrix}, \tag{8}$$

where $\boldsymbol{B}$ is the partial derivative of the first term in the objective function with respect to $\boldsymbol{D}$, $\boldsymbol{A} = \boldsymbol{n}$, $\boldsymbol{D}^*$ and $\boldsymbol{\lambda}^*$ are the optimal displacements and associated Lagrange multiplier, $\boldsymbol{b}$ is the coefficient of the linear term and $\boldsymbol{c} = \boldsymbol{0}$.

After updating the vertices positions with the optimal solution, we project the vertices down to the outer surface of the model by iterating over all the triangles and quads in the mesh to find the closed point to the given updated vertex point. The point that yields the smallest distance from a triangle to a point is not hard to find analytically [34], while we have to apply a newton search to find that for a quad. Since we assume that vertices move along the tangential plane within a reasonably small range can still be considered 'close' to the real geometry, the regularization term $\boldsymbol{w}$ is set to be large and the vertices are not allowed to go far during each position iteration. Therefore, we do multiple times of smoothing optimization steps at a time, which helps us to preserve the geometry while moving the vertices in a decent amount.

V. Results and discussions

To test the robustness of our algorithm, we do several tests on the model of OWN-06c Transonic Airliner provided by NASA from OpenVSP[†], and the eVTOL eCRM-002[‡]. For convenience to add members on the aircraft wing, we remove the turbine engines of the OWN-06c aircraft and all internal members, landing gears and two inner lift nacelles from the eCRM-002 thus the projection algorithm works better with no obstacles.

First of all, we test the projection algorithm, and both ways work well on the OWN-06c aircraft except that finding the closest point does not behave perfectly when we create multiple internal members on the aircraft wing, some members may be very close to the fuselage, if the given projection points are not close enough to the wing surface, the projected points may locate on the fuselage instead of the wing. The reconstruction of the triangulation also behaves nicely on the the OWN-06c aircraft. The intersection curves between the internal members and the aircraft skin seem to match up perfectly, as shown in Fig.7. Also, the intersections of all internal members are detected successfully, and all intersected members are split as we expect.

The input for our algorithm is the surface triangular mesh of the aircraft. Although only one group of splitting, merging, and smoothing optimizations is done on the OWN-06c aircraft skin mesh, the mesh is still transformed into high-quality unstructured quadrilateral mesh. However, the aircraft skin mesh near the intersection curve between the imprinted members and the skin seems to have a lower quality. This is due to the fact that after the reconstruction of triangulation, the region near the intersection tend to have smaller and less regular triangles. The retriangulation algorithm simply connects the projected points to the three vertices of the triangle and creates three new triangles if the projected point locates on the interior of a triangle. Assuming that all the triangles in the original mesh are perfectly equal-lateral, the created triangles can only be less regular and smaller than the triangle before splitting. In addition, since some of members are close to each other, if the average size of the initial mesh is not small enough, the retriangulation algorithm only makes the new triangulation worse. In the extreme case, if the initial triangles are so big that all the projected points of two different members are projected onto the same triangle, the mesh quality improvement methods cannot help a lot with intersection curves fixed.

[†] http://hangar.openvsp.org/vspfiles/75OWN-06cTransonicAirliner

[‡] https://www.uber.com/us/en/elevate/uberair/

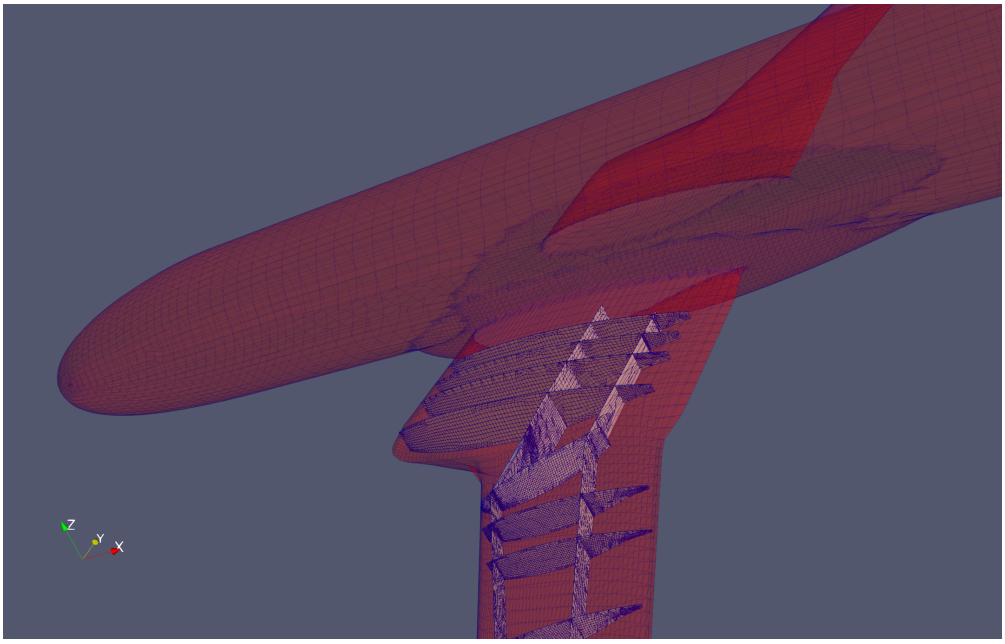


Fig. 7 Test results on an aircraft: The model is the OWN-06c Transonic Airliner from OpenVSP with the turbine engines removed. We create 12 ribs and 4 spars to test the robustness of our algorithm.

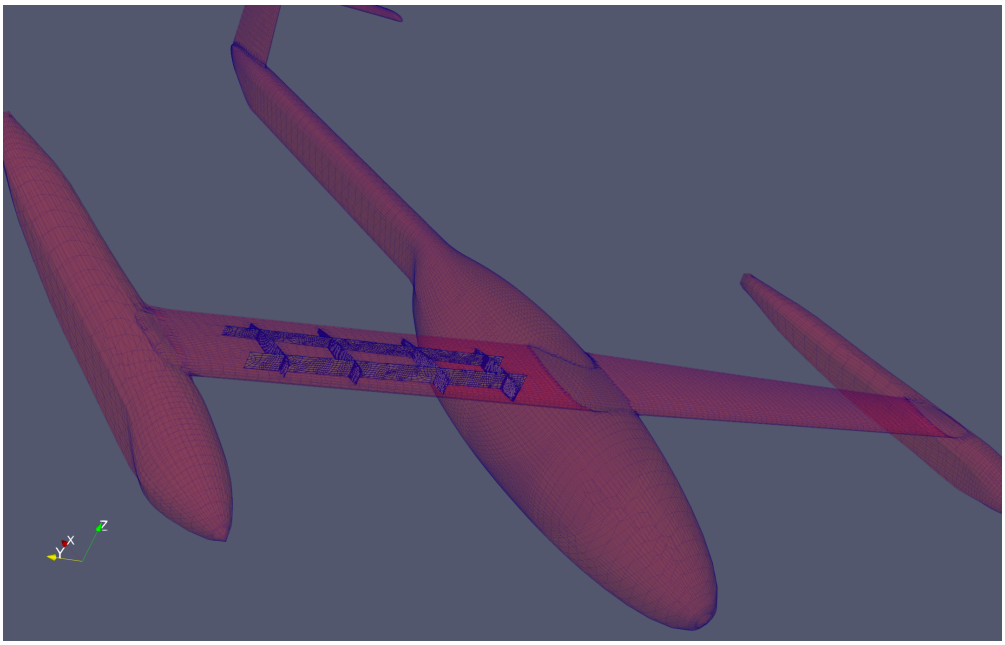


Fig. 8 Test results on an eVTOL: The model is the eVTOL eCRM-002 from Uber with all internal members, landing gears and two inner lift nacelles removed. We create 2 ribs and 4 spars to test the robustness of our algorithm.

As for the mesh of the internal members, we implement a sequence of steps in the mesh quality improvement methods. It is clear that in Fig. 7, for the ribs, the mesh quality improvement methods tend to build structured-like elements in the center, while at the boundary of these imprinted members even though the elements are less regular, the overall quality is considered good. One possible reason for this phenomenon is that points at the boundary of each imprinted member are created by the projection algorithm, which relays significantly on the quality of the initial mesh from the CAD software. Also, the boundary points are fixed throughout the algorithm, which makes them not able to move to improve the quality of elements that contain them. On the other hand, the points at the center are evenly spaced on the parametric domain of a smooth surface created by the TFI algorithm and are not fixed throughout the optimization. With high-quality initial mesh and the ability to move, it is easier for the center regions to end up with structured mesh.

The best thing of this algorithm is that it is fairly efficient, for the mesh generation of the OWN-06c aircraft model with around 9k vertices and 17k triangles, it only takes around 5 minutes to finish, which shows great potential for structural analysis in the conceptual design phase.

However, this algorithm does not work nicely on the OWN-06c eVTOL. The reconstruction of the triangulation gets into trouble when the shape of the object which we project points onto is kind of 'flat' like the wings shown in Fig. 8. In the chordwise direction, the wing of the eVTOL is almost flat, thus in the reconstruction of the triangulation step, we connect the projected points and project the connected lines back. There are no or few intersections detected even though we increase the number of projection points. The retriangulation algorithm can be improved by taking account the normal vector of each triangle. Instead of projecting the connected line backward in the opposite direction we do the projection, we can project all connected lines directly to each triangle, which has a normal vector same direction as the normal vector of the projected points and find the intersection between the projected line and triangulation. Since the reconstruction of the triangulation as the first step does not work well, the rest of the algorithm fails unexpectedly.

VI. Conclusion

In this paper, we presented a novel algorithm to generate shell element meshes for large-scale structural design optimization. Taking an arbitrary triangulation from commercial CAD software, we first define two projection algorithms to project line segments onto the model skin. Then, we reconstruct the triangulation by connecting the projected points and finding the intersection between the projected points and the initial triangulation. Based on the projected points, we can fit curves and create a smooth four-sided surface using transfinite interpolation. The surface can then be transformed into a triangular mesh by evenly interpolating points on the parametric domain and applying a Delaunay triangulation algorithm. Finally, the mesh quality of the imprinted members and the skin of the model is improved after a series of splitting, merging, and smoothing optimizations. There are two major contributions in this paper.

The first contribution is that a complete algorithm is established to generate the smooth quadrilateral meshes for an aircraft taking any arbitrary meshes as input. The algorithm is simplified to only use simple and robust linear and quadratic programming. The algorithm also works for geometries other than the aircraft. For instance, we can apply it to the mesh generation of vehicles, robots, medical devices, etc.

Initially, the smoothing optimization problem is designed to minimize the sum of lengths of all edges, which is a nonlinear programming problem with a high-order solution space. By applying the trust region approach and projection algorithm, the problem is simplified to a iterative quadratic programming optimization problem, which guarantees the robustness.

In summary, the overall approach is potentially a useful way to generate shell element meshes and add imprinted members to the structures with complicated geometry. Starting from our current algorithm, our future work will be adapting the retriangulation algorithm and in the current framework searching for any possibility to make it more general, user-friendly and robust. We will also test the algorithm on other structures from the other fields other than the aerospace field and structures with more complicated geometry. In addition, we want to extend the ideas in this algorithm to the generation of high-quality solid element meshes.

VII. Acknowledgements

We would like to acknowledge the invaluable advice from Professor Kamensky. Also invaluable was the extensive feedback provided by Jiayao Yan, Han Zhao, Ru Xiang. We also feel very grateful for Aobo Yang, Juefan Xie, and Cheng Qian for their suggestions on writing the paper.

References

[1] Raymer, D., *Aircraft design: a conceptual approach*, American Institute of Aeronautics and Astronautics, Inc., 2012.

[2] Jakob, W., Tarini, M., Panozzo, D., and Sorkine-Hornung, O., "Instant field-aligned meshes." *ACM Trans. Graph.*, Vol. 34, No. 6, 2015, pp. 189–1.

[3] Hwang, J. T., and Martins, J. R., "An unstructured quadrilateral mesh generation algorithm for aircraft structures," *Aerospace Science and Technology*, Vol. 59, 2016, pp. 172–182.

[4] Hwang, J., and Martins, J., "GeoMACH: geometry-centric MDAO of aircraft configurations with high fidelity," *12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 2012, p. 5605.

[5] Hahn, A., "Vehicle sketch pad: a parametric geometry modeler for conceptual aircraft design," *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2010, p. 657.

[6] Eriksson, L.-E., "Practical three-dimensional mesh generation using transfinite interpolation," *SIAM journal on scientific and statistical computing*, Vol. 6, No. 3, 1985, pp. 712–741.

[7] Ray, N., Vallet, B., Li, W. C., and Lévy, B., "N-symmetry direction field design," *ACM Transactions on Graphics (TOG)*, Vol. 27, No. 2, 2008, p. 10.

[8] Knöppel, F., Crane, K., Pinkall, U., and Schröder, P., "Globally optimal direction fields," *ACM Transactions on Graphics (ToG)*, Vol. 32, No. 4, 2013, p. 59.

[9] Knöppel, F., Crane, K., Pinkall, U., and Schröder, P., "Stripe patterns on surfaces," *ACM Transactions on Graphics (TOG)*, Vol. 34, No. 4, 2015, p. 39.

[10] Huang, J., Zhou, Y., Niessner, M., Shewchuk, J. R., and Guibas, L. J., "Quadriflow: A scalable and robust method for quadrangulation," *Computer Graphics Forum*, Vol. 37, Wiley Online Library, 2018, pp. 147–160.

[11] Parthasarathy, V., and Kodiyalam, S., "A constrained optimization approach to finite element mesh smoothing," *Finite Elements in Analysis and Design*, Vol. 9, No. 4, 1991, pp. 309–320.

[12] Canann, S. A., Muthukrishnan, S., and Phillips, R., "Topological improvement procedures for quadrilateral finite element meshes," *Engineering with Computers*, Vol. 14, No. 2, 1998, pp. 168–177.

[13] Kälberer, F., Nieser, M., and Polthier, K., "Quadcover-surface parameterization using branched coverings," *Computer graphics forum*, Vol. 26, Wiley Online Library, 2007, pp. 375–384.

[14] Beaufort, P.-A., Geuzaine, C., and Remacle, J.-F., "Automatic surface mesh generation for discrete models: A complete and automatic pipeline based on reparameterization," *arXiv preprint arXiv:2001.02542*, 2020.

[15] Guo, J., Ding, F., Jia, X., and Yan, D.-M., "Automatic and high-quality surface mesh generation for CAD models," *Computer-Aided Design*, Vol. 109, 2019, pp. 49–59.

[16] Verma, C. S., and Tautges, T., "Jaal: Engineering a high quality all-quadrilateral mesh generator," *Proceedings of the 20th International Meshing Roundtable*, Springer, 2011, pp. 511–530.

[17] Lee, Y., Lim, C. K., Ghazialam, H., Vardhan, H., and Eklund, E., "Surface mesh generation for dirty geometries by the Cartesian shrink-wrapping technique," *Engineering with Computers*, Vol. 26, No. 4, 2010, pp. 377–390.

[18] Villard, B., Grau, V., and Zacur, E., "Surface mesh reconstruction from cardiac mri contours," *Journal of Imaging*, Vol. 4, No. 1, 2018, p. 16.

[19] Zhao, D., Chen, J., Zheng, Y., Huang, Z., and Zheng, J., "Fine-grained parallel algorithm for unstructured surface mesh generation," *Computers & Structures*, Vol. 154, 2015, pp. 177–191.

[20] Miranda, A. C., and Martha, L. F., "Mesh generation on high-curvature surfaces based on a background quadtree structure," *space*, Vol. 500, 2002, p. N2.

[21] Park, C., Noh, J.-S., Jang, I.-S., and Kang, J. M., "A new automated scheme of quadrilateral mesh generation for randomly distributed line constraints," *Computer-Aided Design*, Vol. 39, No. 4, 2007, pp. 258–267.

[22] Morfa, C. A. R., Morales, I. P. P., de Farias, M. M., de Navarra, E. O. I., Valera, R. R., and Casañas, H. D.-G., "General advancing front packing algorithm for the discrete element method," *Computational Particle Mechanics*, Vol. 5, No. 1, 2018, pp. 13–33.

[23] Dang, H., Zhang, H., and Dang, H., "Automatic Generation of CFD Mesh for Mountainous Regions," 2017.

[24] Costenoble, A. B., Govindarajan, B., Jung, Y. S., and Baeder, J. D., "Automated Mesh Generation and Solution Analysis of Arbitrary Airfoil Geometries," *23rd AIAA Computational Fluid Dynamics Conference*, 2017, p. 3452.

[25] Singh, J., and Ollivier Gooch, C. F., "Advancing Layer Surface Mesh Generation," *AIAA Scitech 2020 Forum*, 2020, p. 0902.

[26] Rouxel-Labbé, M., Wintraecken, M., and Boissonnat, J.-D., "Discretized riemannian delaunay triangulations," *Procedia engineering*, Vol. 163, 2016, pp. 97–109.

[27] Engwirda, D., and Ivers, D., "Off-centre Steiner points for Delaunay-refinement on curved surfaces," *Computer-Aided Design*, Vol. 72, 2016, pp. 157–171.

[28] Otoguro, Y., Takizawa, K., and Tezduyar, T. E., "A general-purpose NURBS mesh generation method for complex geometries," *Frontiers in Computational Fluid-Structure Interaction and Flow Simulation*, Springer, 2018, pp. 399–434.

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

[30] Tsolakis, C., Drakopoulos, F., and Chrisochoides, N. P., "Sequential metric-based adaptive mesh generation," 2018.

[31] Chen, W., Zheng, X., Ke, J., Lei, N., Luo, Z., and Gu, X., "Quadrilateral mesh generation I: Metric based method," *Computer Methods in Applied Mechanics and Engineering*, Vol. 356, 2019, pp. 652–668.

[32] Soni, A., and Bhowmick, P., "Quadrangular Mesh Generation Using Centroidal Voronoi Tessellation on Voxelized Surface," *International Workshop on Combinatorial Image Analysis*, Springer, 2018, pp. 97–111.

[33] Altenhofen, C., Schuwirth, F., Stork, A., and Fellner, D. W., "Implicit Mesh Generation using Volumetric Subdivision." *VRIPHYS*, 2017, pp. 9–19.

[34] Eberly, D., "Distance between point and triangle in 3D," *Magic Software, http://www. magic-software. com/Documentation/pt3tri3. pdf*, 1999.