Tetrahedral Mesh Improvement by Shell Transformation

Jianjun Chen[a*], Jianjing Zheng[a], Yao Zheng[a], Zhoufang Xiao[a], Hang Si[b], Yufeng Yao[c]

[a] *Center for Engineering and Scientific Computation, and School of Aeronautics and Astronautics, Zhejiang University, Hangzhou 310027, China*
[b] *Weierstrass Institute for Applied Analysis and Stochastics, Mohrenstrasse 39, 10117 Berlin, Germany*
[c] *Faculty of Environment and Technology, University of the West of England, Bristol BS16 1QY, United Kingdom*

ABSTRACT

Existing flips for tetrahedral meshes simply make a selection from a few possible configurations within a single shell (i.e., a polyhedron that can be filled up with a mesh composed of a set of elements that meet each other at one edge), and their effectiveness is usually confined. A new topological operation for tetrahedral meshes named *shell transformation* is proposed. Its recursive callings execute a sequence of shell transformations on neighboring shells, acting like *composite edge removal* transformations. Such topological transformations are able to perform on a much larger element set than that of a single flip, thereby leading the way towards a better local optimum solution. Hence, a new mesh improvement algorithm is developed by combining this recursive scheme with other schemes, including smoothing, point insertion and point suppression. Numerical experiments reveal that the proposed algorithm can well balance some stringent and yet sometimes even conflict requirements of mesh improvement, i.e., resulting in high-quality meshes and reducing computing time at the same time. Therefore, it can be used for mesh quality improvement tasks involving millions of elements, in which it is essential not only to generate high-quality meshes, but also to reduce total computational time for mesh improvement.

KEY WORDS: mesh improvement; mesh generation; shell transformation; mesh smoothing; topological transformation; tetrahedral meshes

1. INTRODUCTION

For numerical simulations with complex geometries, mesh generation typically represents a large portion of the overall computational time. Thus, the ability of performing computations on large-scale tetrahedral elements has always been regarded as an important issue. The fundamental reason is mainly because a theoretically valid tetrahedral mesh can always be automatically generated for a valid 3D domain **[1-5]**, despite that this is not always the case for other specific types of volume elements. Despite of the validity, the quality of an initial tetrahedral mesh produced by a mesher may not be high enough for simulations. A follow-up mesh improvement step is thus indispensable to remove those poorly shaped elements contained in the initial meshes to prevent their adverse effects on the stability and accuracy of the simulations.

In general, a mesh improver executes the following types of local operations iteratively:

(1) *Smoothing*, which repositions mesh points to improve the quality of adjacent elements.
(2) *Local reconnection*, which replaces a local mesh with another mesh that fills up the same region. The new mesh will have the same point set as the old mesh but applying different point connections.

[*] Corresponding author. E-mail: chenjj@zju.edu.cn

(3) *Point insertion/suppression*, which improves a mesh by inserting new points into the mesh or removing existing points from the mesh.

Our primary focus in this study is on local reconnection, although all the local operations mentioned above will be combined in the developed mesh improver. If the point set is fixed, the quality of mesh elements is apparently determined by how these points are connected. It is unrealistic to search for a global optimal mesh topology by directly iterating a large number of possible solutions to connect a point set because this number could expand exponentially with the increase of the number of points. Thus, heuristics prevail in improving the quality of a mesh by iteratively changing the local connections of points.

The most frequently used local reconnection technique for tetrahedral meshes is based on *elementary flips* **[6]**, including 2-3, 3-2 and 4-4 flips (note that the numbers in these names denote the number of tetrahedra removed and created by the flips, respectively; see Figures 1a and 1b). Because the elementary flips simply make a selection from several possible configurations within a relatively small region, their effectiveness in mesh quality improvement is usually confined. To overcome this limit, three advanced flips that involve more elements were later suggested, i.e., *edge removal* **[7]**, *multi-face removal* **[8]** and *multi-face retriangulation* **[9]** (see Figure 1c). They enrich the possible configurations within relatively larger regions and therefore behave more effectively in mesh quality improvement than the elementary flips.

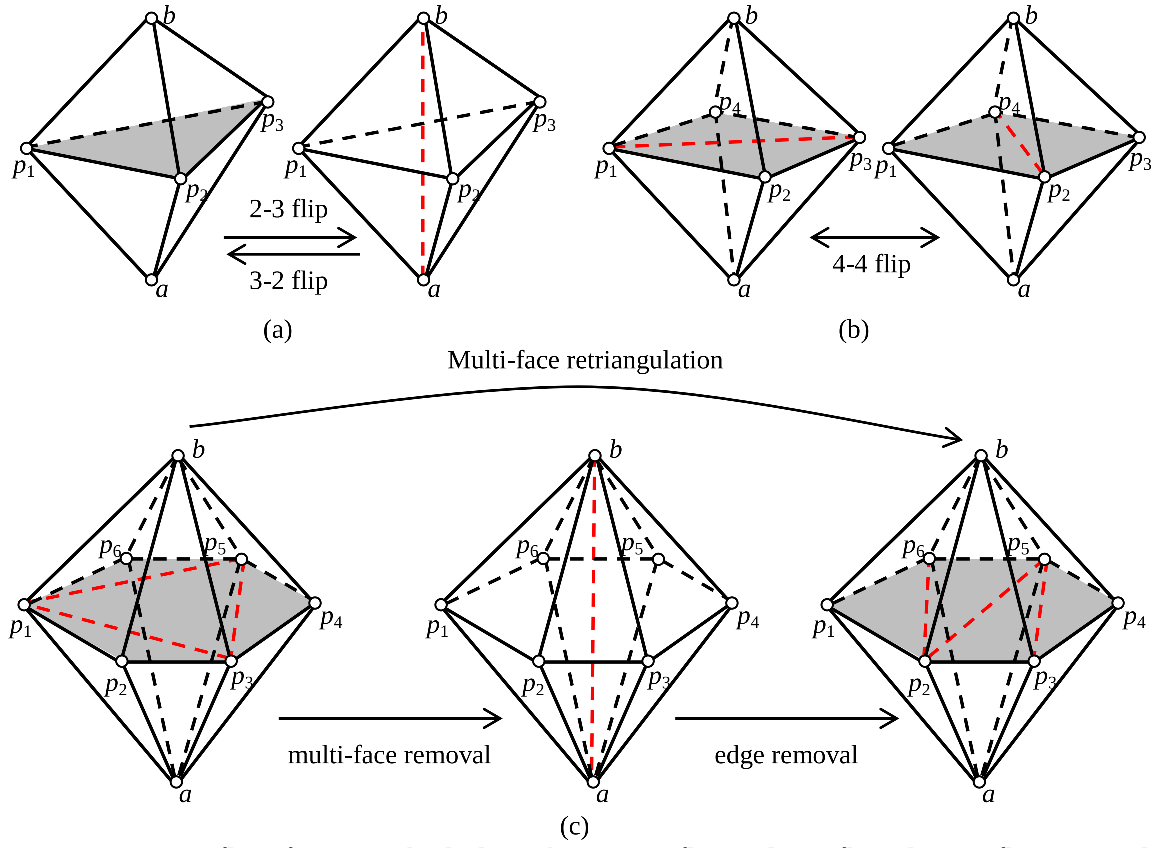


Figure 1. Existing flips for a tetrahedral mesh: (a) 2-3 flip and 3-2 flip; (b) 4-4 flip; (c) multi-face removal, edge removal and multi-face retriangulation.

In general, the effect of a local reconnection technique highly depends on the size of a local mesh it treats. The more elements it treats during one operation; the more possibility it improves the quality of a local mesh to a higher level. This motivates the development of more aggressive local reconnection techniques for mesh quality improvement. For instance, Joe **[6]** once proposed nine schemes to *combine the elementary flips* for mesh improvement. Later, Shewchuk **[10]** pointed out that the composite flips proposed by Joe **[6]** could be expressed as one or two edge removal operations. Thus, Shewchuk suggested that the study of composite edge removal transformations was a fruitful direction for mesh improvement research. Unfortunately, Shewchuk did not present details about what such composite transformations are and how to implement such transformations efficiently. As a result, it was observed that no composite transformations are actually incorporated into the open-source tetrahedral improver (namely Stellar thereafter) developed by Klinger and Shewchuk **[11, 12]**.

The main contribution of this present study is the development of a new local reconnection technique that could act like composite edge removal transformations. This technique is based on the recursive callings of a new flip named *shell transformation*. The single calling of shell transformation could be considered as an enhanced version of edge removal transformation. However, an essential difference exists between two approaches, and enables shell transformation to be executed recursively (see Section 2.4 for details). Thus, edge removal only makes a selection from a few possible configurations within a single shell (i.e., a polyhedron that can be filled up with a mesh composed of a set of elements that meet each other at one common edge). However, the recursive callings of shell transformations can execute a sequence of shell transformations on neighboring shells. In other words, recursive shell transformations could be performed on a much larger element set than that of edge removal, thereby leading the way towards a better local optimum solution.

Another focus of this study is about the efficient implementation of the new local reconnection technique, because local reconnections need to be employed for a large number of times during the entire mesh improvement workflow. The dynamic programming algorithm suggested by Shewchuk for edge removal **[10, 13]** will be revisited at first and then further enhanced to implement the basic shell transformation routine. Besides, the computing efficiency of recursive callings of shell transformations is investigated carefully because the number of such callings may increase exponentially when the recursive level increases. Reasonable restrictions are provided to prevent inefficient recursive callings. Meanwhile, several strategies are suggested to improve the efficiency.

Finally, the ability of the proposed local reconnection technique will be demonstrated by performing various mesh improvement tasks, some of which involve millions of elements. In a pipeline of producing meshes of this magnitude, mesh generation itself may only consume a few seconds computing time, owing to recent advancement in the field of fast and parallel mesh generation techniques **[14-17]**. However, a mesh improver possibly consumes many minutes computing time or even longer in order to manage such a big mesh. Therefore, to ensure the applicability of this newly developed mesh improver for large-scale problems, an essential requirement we will take into account is the *cost-effectiveness* of the mesh improver, i.e., the ability to balance the conflict requirements of resulting in a high-quality mesh and saving computing time of mesh improvement. Following this concept, a set of existing smoothing, point insertion and point suppression schemes are selected. Combining these schemes with the proposed local reconnection technique, a cost-effective improver applicable to large-scale meshes is therefore developed and verified.

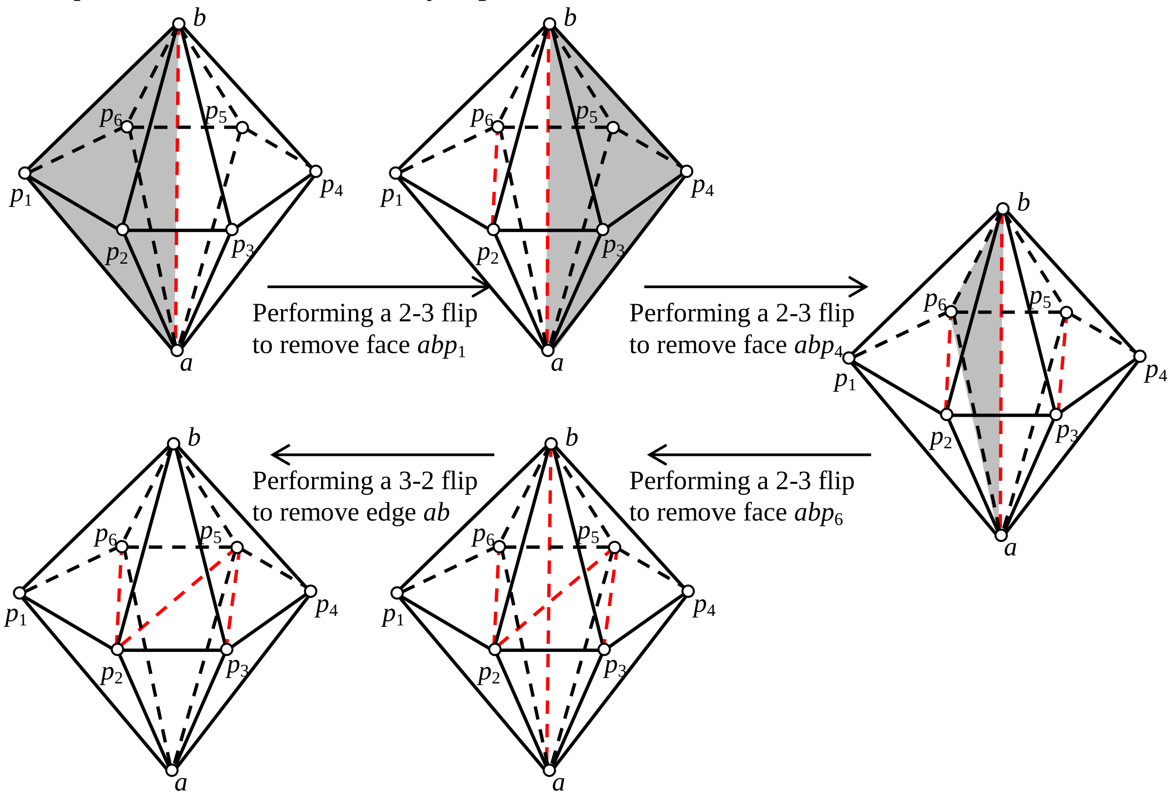
The remainder of this article will be organized as follows. In Section 2, related works are firstly reviewed, followed by the introduction of the new local reconnection technique in Section 3. Sections 4 describes the basic implementation of shell transformation, while Section 5 presents the recursive scheme of shell transformation and the local reconnection scheme based on this recursive scheme. Section 6 introduces other local operations that are combined to form the developed new mesh improver. Section 7 provides various examples of numerical experiments demonstrating the effectiveness and efficiency of the proposed scheme. Section 8 concludes with outcomes of the study.

2. RELATED WORKS

Firstly, related works on local reconnection techniques are reviewed in details (see Section 2.1). After that, a brief review on other types of local operations (see Sections 2.2 and 2.3, respectively) is presented to justify our choices of these types of operations in the developed mesh improver.

2.1 Related work on local reconnection techniques

Local reconnection techniques are frequently used in various circumstances of mesh generation, such as Delaunay refinement **[18]**, mesh adaptation **[19]**, boundary recovery **[1-5]**, and mesh quality improvement **[6-12, 20, 21]**. The first type of local reconnection techniques is based on the flips presented in Figure 1. The 3-2, 2-3 and 4-4 flips are defined as *elementary flips*, not only because they are special cases of the advanced flips, but also because they could be combined to form the advanced flips. For instance, the edge removal transformation could be implemented as a sequence of 2-3 flips followed by a single 3-2 flip (see Figure 2) **[10]**. This possibly explains why Shewchuk **[10]** judged that the composite transformations of the elementary flips proposed earlier by Joe **[6]** could be expressed as one or two edge removal transformations. However, this does not mean that a local reconnection technique based on the elementary flips could achieve the same effect as the techniques based on the advanced flips. In general, the advanced flips provide elaborate '*patterns*' for how to combine the elementary flips such that those involved elementary flips could work together in a much larger element set. As a result, the local reconnection techniques based on the advanced flips could usually improve the quality of a mesh to a higher level than that by the techniques based on the elementary flips.



b
p6
p5
p1
p4
p2
p3
a
Performing a 2-3 flip to remove face abp_1
Performing a 2-3 flip to remove face abp_4
Performing a 2-3 flip to remove face abp_6
Performing a 3-2 flip to remove edge ab

Figure 2. Implementing edge removal as a sequence of 2-3 flips followed by a 3-2 flip.

Differently, Liu *et al.* proposed to improve mesh quality by a local remeshing technique named *small polyhedron reconnection* (SPR) **[21]**. The SPR algorithm initializes an empty polyhedron in the neighbourhood of a poorly shaped element, and then performs an exhaustive search to find the *optimal* tetrahedralization of this polyhedron. Typically, the advanced flips treat only a local mesh composed of a few tetrahedral elements, while the SPR algorithm can search for the optimal tetrahedralization of a polyhedron composed of 20-40 tetrahedral elements. Consequently, it was reported that the SPR algorithm could achieve a better mesh improvement result than that of the flip-based algorithm **[21]**. However, the main issue of the SPR algorithm is its computing complexity, since the problem of meshing an empty polyhedron is NP hard. Although some strategies have been proposed in the past to improve the timing-related performance of the SPR algorithm **[4, 21]**, our experience shows that, if the local reconnection scheme of a mesh improver is completely dependent on the SPR algorithm. In addition, the runtime of a mesh improver may be beyond the user's expectation, in particular when the treated mesh is composed of one million or more elements.

2.2 Related work on mesh smoothing

The mesh smoothing methods can be classified into two categories in general, i.e., the Laplacian-type method **[22]** and the optimization-based method **[20, 23-28]**. The Laplacian-type method moves each mesh vertex towards the *center* of its neighboring vertices. It provides no guarantee for the improvement of mesh quality, therefore leading to either low quality or even possibly invalid elements. To overcome this drawback, various optimization-based approaches were proposed. These approaches can be divided into two types in accordance with the choice of objective functions. The *local* method maximizes a quality function for the elements surrounding each mesh vertex by relocating mesh vertices iteratively **[20, 23, 24]**. The *global* method maximizes a quality function for all elements by relocating all mesh vertices simultaneously **[25-28]**. While the local method may improve an element at a cost of degrading its neighboring elements, the global method relieves this conflict to some extents by considering the quality of the entire mesh as a whole. Nevertheless, because the global method needs to solve a large-scale optimization model, the choice of solution methods for this optimization model will be the key step to achieve acceptable computing time and performance **[28]**. Likewise, in the case of surface mesh smoothing, various *local* and/or *global* shape-preserving approaches have been developed, in which the global approach based on geometric flows is now still prevailing because of its powerful ability to preserve geometric features and thus reduce volume shrinkage **[25]**.

Since the cost-effectiveness of a new mesh improver is our primary goal to achieve, we select a local approach instead of a more time-consuming global approach for mesh smoothing. More specifically, we combine an optimization-based algorithm **[20]** with the Laplacian smoothing **[22]**. See Section 5.4 later for more details.

2.3 Related work on point insertion and point suppression

It is intuitive to eliminate low-quality elements by inserting vertices into existing meshes. Typically, a new vertex can be located at the circumcenter **[5, 29]** or centroid of a poor element **[1, 4]**, or the midpoint of the longest edge of this element **[2, 30]**. One typical strategy to insert this kind of vertex into a mesh is by Delaunay refinement **[1-5, 29, 31]**, or more simply, by splitting the element(s) containing the new vertex (which may result in temporary low-quality elements) firstly and then improving the mesh by combining smoothing and local reconnection **[19]**. Besides, Klingner and Shewchuk suggested an effective but very time-consuming point insertion scheme that combines a Delaunay-type algorithm with smoothing operations **[11]**. It is possible that hundreds of elements are involved in just one single operation. Nevertheless, to meet our goal of developing a cost-effective mesh improver for large-scale problem inputs, we adopt an *edge-splitting* based point insertion scheme.

As a reverse operation of point insertion, it is not surprising that point suppression can also improve local mesh quality by remeshing an empty polyhedron composed of all elements surrounding the vertex to be removed. The challenge mainly comes from those polyhedra that cannot be tetrahedralized if no Steiner points are allowed **[31]**. Meanwhile, even if a polyhedron can be tetrahedralized without inserting Steiner points (although the prediction of this is NP hard **[31]**), it is not easy to find an optimal mesh to fill in that polyhedron (it is also NP hard **[4, 21]**). Theoretically, the SPR algorithm mentioned in Section 2.1 **[4, 21]** could be a good candidate for remeshing an empty polyhedron because it can provide an optimal solution when the polyhedron is meshable. Nevertheless, as we pointed earlier, the main issue of the SPR algorithm is its relatively poor timing performance. Therefore, we adopt an *edge-contraction* based point suppression routine that is available from Stellar **[12, 13]**.

See Section 5.5 later for the developed point insertion and suppression schemes.

3. SHELL TRANSFORMATION AND ITS RECURSIVE CALLINGS: THE MAIN IDEA

For the completeness, we have briefly reviewed different types of local operations in order to justify our choices. However, it must be emphasized that the main contribution of this study is the development of a new local reconnection technique. With respect to other local operations, we only select a suitable operation among various existing approaches to meet our goal of developing a cost-effective mesh improver.

Before introducing the new local reconnection technique, Figure 3 illustrates some terminologies in relation with a shell structure because all of the flips discussed in this study, including those presented earlier in Figure 1 and the new proposed shell transformation, will involve this structure.

It was reported that edge removal might be the most effective flip for mesh quality improvement **[12]**, although other flips could provide marginal improvements additionally. To remove a low-quality element (for instance, the element abp_5p_6 in Figure 4), we could pick up an edge of this element (denoted by e), and perform the edge removal transformation in the shell of e (i.e., e is the supporting edge of the shell). For instance, e refers to ab in Figure 4. If the output covering mesh of the transformation is better than the old one, edge removal succeeds and the low-quality element is removed (since e is removed). In such a case, the skirt polygon of the shell is *completely triangulated* (see Figure 4). However, it is not uncommon that the edge removal could not provide a better mesh than the old one and thus fails to remove e and the low-quality element bounded by e. Instead, a covering mesh of the shell that is better than the old one might be the one shown in the bottom of Figure 4, where a *core* referring to the unmeshed part of the skirt polygon exists in the resulting mesh. In this case, we say the shell of e can only be partially reduced[†], and the remaining supporting faces must be removed to reduce the shell further. Obviously, if one of the link edges that bound a supporting face f is removed, f will be removed accordingly. For the case shown in the bottom of Figure 4, f could be the face abp_5, and the link edge could be ap_5 or bp_5. Assuming that ap_5 is picked up for removal, the above transformation is called again to reduce the shell of ap_5. If the reduced shell of ap_5 does not contain the face abp_5 and any new supporting faces sharing

[†]Here, the *degree* of a covering mesh refers to the number of elements that share the supporting edge in this mesh. A shell is *reduced* if the degree of the new covering mesh is becoming smaller than that of the old mesh. In particular, if the degree of the new mesh becomes zero, the shell is *completely reduced,* and the new mesh is a *completely reduced mesh*; otherwise, the shell is *partially reduced,* and the new mesh is a *partially reduced mesh*.

ab, the shell of *ab* is reduced as well; otherwise, a process that attempts to remove the supporting faces around ap_5 is repeated.

To better understand the above recursive scheme, Figure 5 illustrates how this scheme works on a local mesh composed of two shells (see Figure 5a), aimed at removing the edge *ab* from the mesh. Firstly, a transformation is called on the shell of *ab*. Since the shell cannot be completely reduced, the edge *ab* still exists in the output mesh (see Figure 5b). Nevertheless, the degree of the shell is reduced from 5 to 4. To reduce the shell further, a link edge *bh* is picked up and a transformation is called on the shell of *bh* and reduces this shell completely. Besides, the degree of the shell of *ab* is reduced from 4 to 3 after this step (see Figure 5c). Finally, a transformation is called to update the shell of *ab* to remove *ab* by a single 3-2 flip (see Figure 5d for the final output).

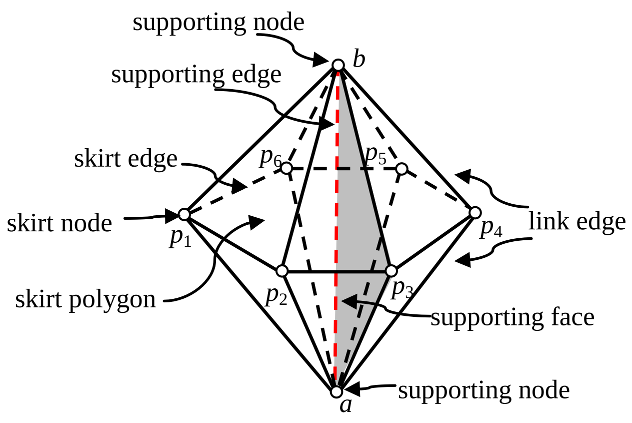


Figure 3. The terms defined for the mesh entities of a shell.

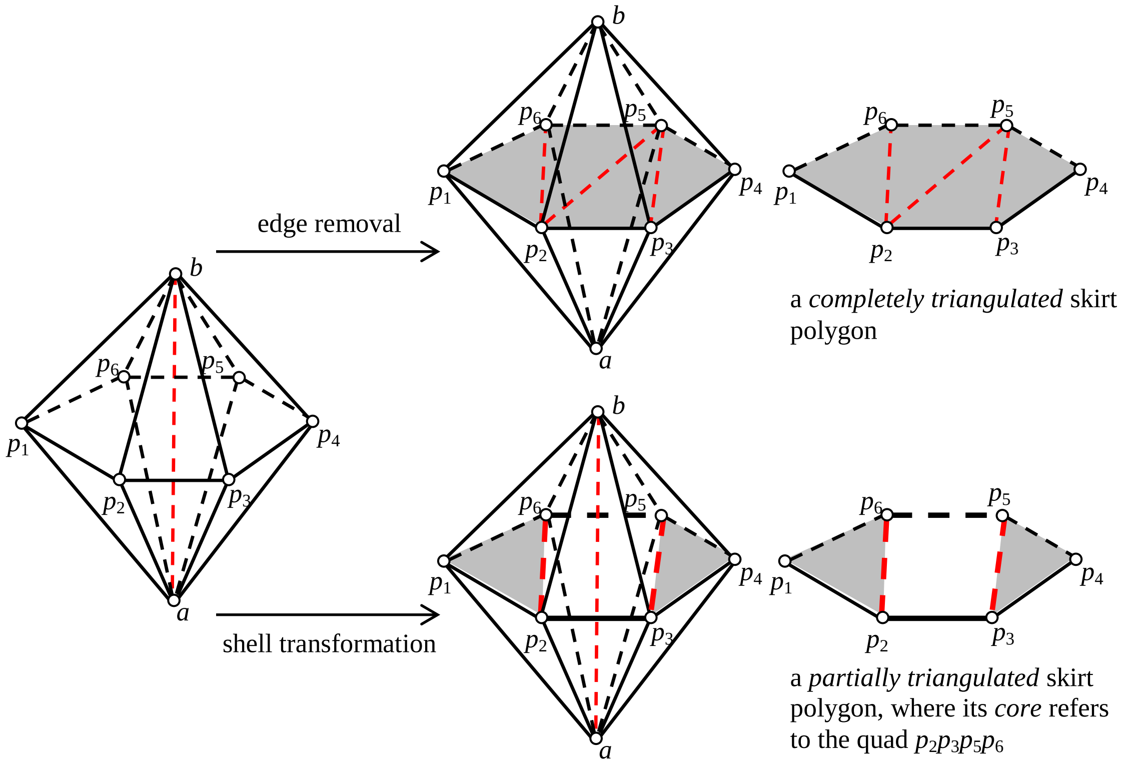

b
p6
p5
p1
p4
p2
p3
a
edge removal
shell transformation
a *completely triangulated* skirt polygon
a *partially triangulated* skirt polygon, where its *core* refers to the quad $p_2p_3p_5p_6$

Figure 4. The difference between edge removal and a single calling of shell transformation. This difference enables shell transformation to be called recursively while edge removal cannot. This recursive ability is the main advantage of shell transformation technique.

Here, given the covering mesh of a shell, the transformation that attempts to reduce the shell is denoted as *shell transformation*. The main *difference* between shell transformation and edge removal is that shell transformation allows the output mesh containing a *partially triangulated* skirt polygon while edge removal does not. If the output covering mesh of shell transformation does not contain a *core*, shell transformation is the same operation as edge removal. In this respect, edge removal could be considered a special case of shell transformation. In other words, shell transformation could be considered as an enhanced version of edge removal because it considers more possibility to mesh a shell. Moreover, *the main advantage of shell transformations is rooted in its recursive ability*. As Joe have demonstrated **[6]** that the composite transformations of elementary flips could improve the quality of a mesh to a much higher level than elementary flips, and the recursive callings of shell transformations, acting like composite edge removal transformations, could perform much better than edge removal in most mesh improvement tasks, as we will demonstrate in Section 6.

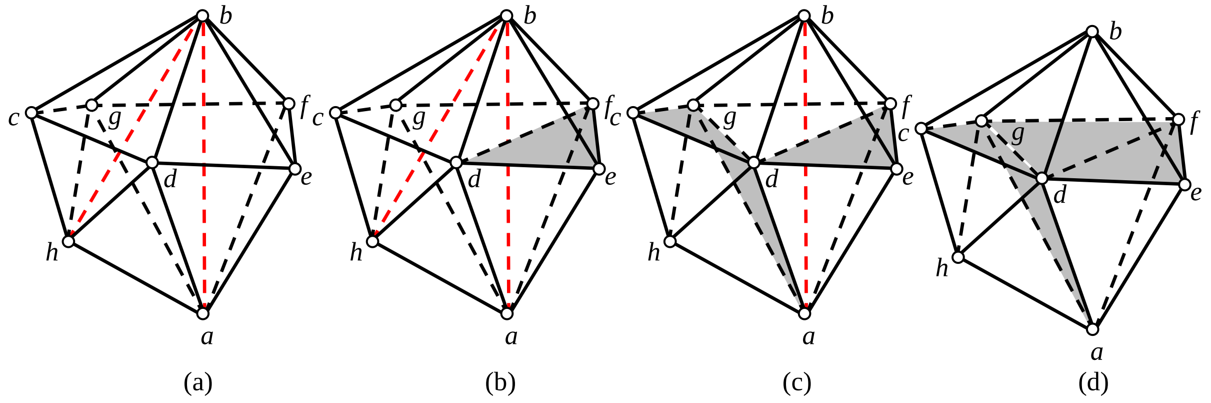


Figure 5. Illustration for the recursive callings of shell transformations. (a) The input mesh. (b) The output after the first shell transformation calling on the shell of *ab*. (c) The output after the second shell transformation calling on the shell of *bh*. (d) The final output after the third shell transformation calling on the shell of *ab*.

4. A SINGLE CALLING OF SHELL TRANSFORMATION

To implement a shell transformation procedure, a key step is to develop an algorithm that can triangulate a skirt polygon *partially*. Meanwhile, among all of the valid triangulation schemes, an algorithm needs to find the triangulation corresponding to an *optimal* covering mesh. In this section, we will first reproduce a dynamic programming algorithm suggested by Shewchuk **[10]**, which can be used to triangulate the skirt polygon *completely* and optimally. Next, the proposed shell transformation algorithm is described in details, which enhances the Shewchuk's algorithm in order to triangulate the skirt polygon *partially* and optimally.

4.1 The algorithm proposed by Shewchuk **[10]**

Given a shell covered by a set of tetrahedra $p_i p_{i+1} ab (i=1,2,\cdots,m)$, each triangulation $\mathrm{T}=\{t_1,t_2,\cdots t_{m-2}\}$ of the skirt polygon induces a tetrahedralization of the shell:

$$\mathrm{K}=\{\mathrm{conv}(t_i,a)\cup\mathrm{conv}(t_i,b)\,|\,i=1,2,\cdots,m-2\}.$$

Here, $\mathrm{conv}(\cdot)$ refers to a tetrahedron formed by a face and a node. We define the quality of T to be the quality of the worst tetrahedron within the tetrahedralization K.

For each node $p_i (1\le i\le m)$, p_{i+m} is its alias. $\mathrm{R}_{i,j}$ defines a ring of edges whose ending nodes are

$$\mathrm{P}_{i,j}=\begin{cases}\{p_i,p_{i+1},\cdots p_j\} & 1\le i<j\le m\\ \{p_j,p_{j+1},\cdots p_{i+m}\} & 1\le j<i\le m\end{cases}.$$

Selecting a node $p_k\in\mathrm{P}_{i,j}$ other than p_i and p_j, the triangulation of $\mathrm{R}_{i,j}$ includes three parts, as shown in Figure 6:

$$\mathrm{T}_{i,j}=\mathrm{T}_{i.k}\cup\mathrm{T}_{k,j}\cup\Delta p_i p_k p_j.$$

We can define a matrix $\mathbf{M}_q$ to record the quality of the optimal triangulation of $\mathrm{R}_{i,j}$ as:

$$\mathbf{M}_q(i,j)=\max_{p_k\in\mathrm{P}_{i,j},p_k\neq p_i,p_j}\min\{\mathbf{M}_q(i,k),\mathbf{M}_q(k,j),q(a,p_i,p_j,p_k),q(p_i,p_j,p_k,b)\},\qquad(1)$$

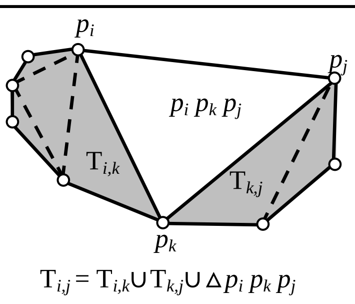
where $q(\cdot)$ is the quality function for tetrahedral elements. $\mathbf{M}_q(i,j)=\infty$ when $j=i+1$. Since $1\le i,j\le m$ (m refers to the size of the skirt polygon), we know $\mathbf{M}_q$ is a $m\times m$ matrix.

Based on Equation 1, Algorithm 1 fills in the upper triangular part of $\mathbf{M}_q$ by an order of decreasing i and increasing j so that $\mathbf{M}_q(i,k)$ and $\mathbf{M}_q(k,j)$ are computed before $\mathbf{M}_q(i,j)$. Meanwhile, Algorithm 1 fills in another matrix $\mathbf{M}_k$ that records the values of k that maximize Equation 1 in order to reconstruct the optimal triangulation $\mathrm{T}^{\mathrm{opt}}$:

$$\begin{cases}\mathrm{T}^{\mathrm{opt}}=\mathrm{T}^{\mathrm{opt}}_{1,m}\\ \mathrm{T}^{\mathrm{opt}}_{i,j}=\mathrm{T}^{\mathrm{opt}}_{i,k}\cup\mathrm{T}^{\mathrm{opt}}_{k,j}\cup\Delta p_i p_k p_j\quad(k=\mathbf{M}_k(i,j))\end{cases}.$$

Algorithm 1. Filling in the upper triangular part of $\mathbf{M}_q$ and $\mathbf{M}_k$

```
fillInMatrices_UpRight($a$, $b$, P, $\mathbf{M}_q$, $\mathbf{M}_k$)
Inputs:
    $a$ and $b$: the support nodes of a shell,
    P = {$p_1$, $p_2$, …, $p_m$}: the skirt polygon of a shell
  1.  for $i = m - 2$ downto 1
  2.      for $j = i + 2$ to $m$
  3.          for $k = i + 1$ to $j - 1$
  4.              $q = \min\{q(a, p_i, p_k, p_j), q(p_i, p_k, p_j, b)\}$
  5.              if $k < j - 1$
  6.                  $q = \min\{q, \mathbf{M}_q(k, j)\}$
  7.              if $k > i + 1$
  8.                  $q = \min\{q, \mathbf{M}_q(i, k)\}$
  9.              if $k = i + 1$ or $q > \mathbf{M}_q(i, j)$
  10.                 $\mathbf{M}_q(i, j) = q$
  11.                 $\mathbf{M}_k(i, j) = k$
```



p_i
p_j
$p_i\, p_k\, p_j$
$\mathrm{T}_{i,k}$
$\mathrm{T}_{k,j}$
p_k
$\mathrm{T}_{i,j} = \mathrm{T}_{i,k}\cup\mathrm{T}_{k,j}\cup\Delta p_i\, p_k\, p_j$

Figure 6. Decomposing a triangulation optimization problem into sub-problems.

4.2 The proposed shell transformation algorithm

Firstly, we introduce the concept of *triangulation graph*. It is a directed graph defined on a polygon that is bounded by a set of nodes $\mathrm{P} = \{p_1, p_2, \cdots p_m\}$. A *graph node* corresponds to a polygon node, and a graph edge $\langle p_i, p_j \rangle$ exists if there are valid triangulations for the ring $\mathrm{R}_{i,j}$. Note that $\mathbf{M}_q$ provides a representation of the triangulation graph: $\mathbf{M}_q(i, j) > 0$ means that there is a valid triangulation for the ring $\mathrm{R}_{i,j}$; thus, $\langle p_i, p_j \rangle$ is a graph edge.

Algorithm 1 only fills in one half of $\mathbf{M}_q$ and $\mathbf{M}_k$. To get a complete representation of the triangulation graph, the lower left elements of $\mathbf{M}_q$ and $\mathbf{M}_k$ must be computed. Algorithm 2 present a routine that fills in all of useful elements of $\mathbf{M}_q$ and $\mathbf{M}_k$, which could be located in either side of the main diagonal of the two matrices. In the new algorithm, one diagonal of $\mathbf{M}_q$ and $\mathbf{M}_k$ is computed at a time in the increasing order of the size of $\mathrm{R}_{i,j}$ (i.e., number of vertices). Note that after calling Algorithm 2, $\mathbf{M}_q(1, m)$, and $\mathbf{M}_q(2,1)$, ……, and $\mathbf{M}_q(m, m-1)$ all records the quality of the optimal triangulation of the complete skirt polygon, but with different start and end vertices. Based on Algorithm 2, we could then implement a single calling of shell transformation (see Algorithm 3), where a key step is to define the *core* of the shell. As shown in Figure 7, the core introduced in shell transformation corresponds to a simple cycle of the triangulation graph of the skirt polygon. Therefore, once the graph is set up by calling Algorithms 1 and 2, all of the simple cycles are visited, and the optimal one is picked up to reconstruct the triangulation of the skirt polygon (i.e., Line 5 of Algorithm 3). Here, a simple cycle is identified as optimal when it corresponds to an optimal covering mesh (K). Note that the definition on the optimality of a mesh could be application specific, see Section 4.3 for details.

Algorithm 2. Filling in $\mathbf{M}_q$ and $\mathbf{M}_k$

```
fillInMatrices($a$, $b$, P, $\mathbf{M}_q$, $\mathbf{M}_k$)
Inputs:
    $a$ and $b$: the support nodes of a shell
    P = {$p_1$, $p_2$, …, $p_m$}: the skirt polygon of a shell
  1.  for $d$ = 2 to $m$ - 1
  2.      for $i$ = 1 to $m$
  3.          $j' = i + d$
  4.          for $k' = i + 1$ to $j' - 1$
  5.              $j = j' > m \ ?\ j' - m : j'$
  6.              $k = k' > m \ ?\ k' - m : k'$
  7.              $q = \min\{q(a, p_i, p_k, p_j), q(p_i, p_k, p_j, b)\}$
  8.              if $k' < j' - 1$
  9.                  $q = \min\{q, \mathbf{M}_q(k, j)\}$
  10.             if $k' > i + 1$
  11.                 $q = \min\{q, \mathbf{M}_q(i, k)\}$
  12.             if $k' = i + 1$ or $q > \mathbf{M}_q(i, j)$
  13.                 $\mathbf{M}_q(i, j) = q$
  14.                 $\mathbf{M}_k(i, j) = k$
```

Assuming that the node set of the core is $\mathrm{P}_c = \{p_{c_1}, \cdots p_{c_n}, p_{c_{n+1}} = p_{c_1}\}$, the reconstructed triangulation is (see Figure 7):

$$\mathrm{T}^{\mathrm{opt}} = \{\mathrm{T}^{\mathrm{opt}}_{c_j, c_{j+1}} \mid j = 1, 2, \cdots, n\}$$

The new covering mesh of the shell is:

$$\begin{aligned} &\mathrm{K} = \mathrm{K}_1 \cup \mathrm{K}_2 \\ &\mathrm{K}_1 = \{\mathrm{conv}(t_i, a) \cup \mathrm{conv}(t_i, b) \mid t_i \in \mathrm{T}^{opt}\}, \\ &\mathrm{K}_2 = \{\mathrm{tetr}(p_{c_j}, p_{c_{j+1}}, a, b) \mid j = 1, 2, \cdots, n\} \end{aligned} \tag{2}$$

where $\mathrm{tetr}(\cdot)$ refers to the tetrahedral element formed by four specified nodes.

Algorithm 3. A general routine of shell transformation

```
shellTransformation($a$, $b$, P, $K_{old}$)
Inputs:
    $a$ and $b$: the support nodes of the shell
    P = {$p_1$, $p_2$, …, $p_m$}: the skirt polygon of the shell
    $K_{old}$: the old covering mesh of the shell
 1. fillInMatrices_UpRight($a$, $b$, P, $\mathbf{M}_q$, $\mathbf{M}_k$)
 2. fillInMatrices_LowLeft($a$, $b$, P, $\mathbf{M}_q$, $\mathbf{M}_k$)
 3. $G$: the triangulation graph with $\mathbf{M}_q$ as its matrix representation
 4. $P_c$: an optimal simple cycle of $G$
 5. $T^{opt}$: the reconstructed triangulation from $P_c$, see Figure 6
 6. K: the new covering mesh of the shell, see Equation 2
 7. if $K \neq \emptyset$ and $K \neq K_{old}$
 8.     Remesh the shell by replacing $K_{old}$ with K
```

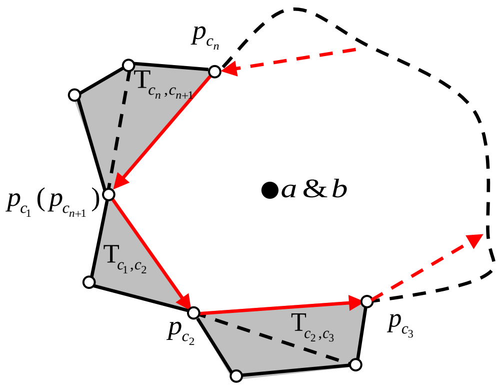


Figure 7. Illustration for the triangulation graph of a polygon, where those lines with arrows are a group of graph edges, forming a simple cycle and thus defining a scheme that triangulates the polygon *partially*.

5. RECURSIVE CALLINGS OF SHELL TRANSFORMATIONS

5.1 The basic routine

A single shell transformation calling only involves a small number of elements. It may not be able to reduce a shell completely because of the constraints on the shell boundaries. Hence, we develop a routine that calls shell transformations recursively to remove these constraints.

Algorithm 4 details the routine of recursive shell transformations. Given an edge e, the calling **recursiveST(**e, $\emptyset$, 0, $l_{\max}$**)** attempts to remove the edge e, where $l_{\max}$ limits the maximally allowed recursive level. Given a face f and one of its boundary edges e, the calling **recursiveST(**e, f, 0, $l_{\max}$**)** attempts to remove the face f.

Note that Algorithm 4 expands an *edge tree*, where the input edge e is the *root* of this tree, and those link edges (e') inputted for further recursions are *children* of the edge e. In this manner, the tree can be expanded recursively. If a tree node v_1 is an *ancestor* of another tree node v_2, we say the edge corresponding to v_1 is an *ancestor edge* of the edge corresponding to v_2.

5.2 Termination and efficiency

Two routines called by Lines 10 and 3 account for the timing-related performance of Algorithm 4, namely **pickRecursiveLinkEdge** and **shellTransformation**. They determine how many shell transformations are executed and how fast a single shell transformation can run, respectively.

Given a supporting face f in the shell of e, the routine **pickRecursiveLinkEdge** checks whether or not a further recursion is necessary. If yes, the routine returns a link edge between two possible candidates. To filter inefficient recursions, the implementation of this routine can be further improved through the following guidelines:

(1) Do not return a boundary edge.

(2) If a tetrahedra sharing f overlaps with the shells of an ancestor edge of e, return nothing.

(3) Return a *reflex edge* only. In Figure 8, two faces ap_2p_3 and ap_3p_4 form a *reflex angle* if viewed from a point b; correspondingly, ap_3 is called a *reflex edge* of the face abp_3.

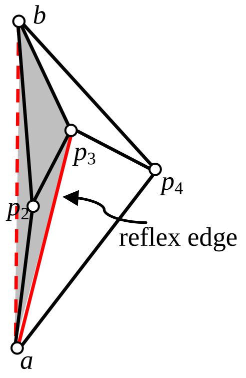


Figure 8. Illustrative case of a *reflex edge*.

The termination of Algorithm 4 remains an issue because there is no guarantee that a mesh edge could be removed by flips under the requirement that the quality of the mesh could not be decreased by these flips. Although the second guideline mentioned above prevents an infinite execution of the recursive callings, Algorithm 4 could still be very time-consuming because the number of its shell transformation callings may increase exponentially when the recursive level increases. Therefore, a user parameter $l_{\max}$ is input to Algorithm 4 to limit the maximal recursive level. Based on an analysis of many trial and error experimental results, we choose $l_{\max}$ to be 5 in this study to meet our goal of developing a cost-effective mesh improver.

The routine **shellTransformation** (see Algorithm 3) includes two main steps. Firstly, it employs Algorithms 1 and 2 to obtain the triangulation graph of the skirt polygon. Next, it searches for a simple cycle in that graph to reconstruct an optimal covering mesh according to Equation 2. The time consumptions of both steps are at an order of $O(m^3)$, where m is the number of skirt nodes. A single calling of this routine may consume little time because m is very small. However, the number of callings could be very large because of the recursive nature of Algorithm 4. In our implementation, this routine has been speeded up through three treatments as follows:

(1) Introduce the validity conditions (see Section 4.3) in the first step to simplify the triangulation graph;
(2) Search for the *optimal* simple cycle first to prevent those unnecessary searches for low-quality cycles;
(3) Record the results of time-consuming mesh validity and quality computations after they are executed for the first time so that simple queries can replace the repeated callings of these computations.

Algorithm 4. The routine of recursive shell transformations

```
recursiveST($e$, $f$, $l$, $l_{\max}$)
Inputs:
    the supporting edge, denoted $e$
    a face containing $e$ that the routine attempts to remove, denoted $f$
    the recursive level with an initial value of zero, denoted $l$
    the maximally allowed recursive level, denoted $l_{\max}$
Variables:
    the ending nodes of an edge, denoted $a(\cdot)$ and $b(\cdot)$
    the skirt polygon of the shell of an edge, denoted $\mathrm{P}(\cdot)$
    the set of elements containing an edge, denoted $\mathrm{S}(\cdot)$
    a set of link faces contained in $\mathrm{S}(\cdot)$, denoted $\mathrm{F}(\cdot) = \{ f_1', f_2', \cdots, f_m'\}$, where $m = |\mathrm{F}(\cdot)|$
 1. if | $\mathrm{S}(e)$| <= 0 or ($f$ != $\varnothing$ and $f \notin \mathrm{F}(\mathrm{S}(e))$)
 2.     return success
 3. shellTransformation($a(e)$, $b(e)$, $\mathrm{P}(e)$, $\mathrm{S}(e)$)
 4. if | $\mathrm{S}(e)$| <= 0 or ($f$ != $\varnothing$ and $f \notin \mathrm{F}(\mathrm{S}(e))$)
 5.     return success
 6. if $l$ >= $l_{\max}$  /* the resursive level is limited under $l_{\max}$. */
 7.     return fail
 8. $m$ = |$\mathrm{S}(e)$ | /* record the size of the shell $\mathrm{S}(e)$ */
 9. for $i$ = 1 to $m$
10.   $e'$ = pickRecursiveLinkEdge($f_i'$) /* filters are set to avoid inefficient recursions */
11.   if $e'$ != $\varnothing$
12.       recursiveST($e'$, $f_i'$, $l$+1, $l_{\max}$) /* recursive calling */
13.       if |$\mathrm{S}(e)$ | < $m$ /* $\mathrm{S}(e)$ is reduced as well */
14.           retrun recursiveST($e$, $f$, $l$, $l_{\max}$) /* recursive calling */
15. return fail
```

Another factor that affects the efficiency of Algorithm 4 is the routine that identifies a shell in the input mesh (referring to the *shell-find* routine thereafter), which is employed by Lines 4 and 13 of Algorithm 4. The efficiency of this routine depends on the data structure adopted to represent a tetrahedral mesh. In our scheme, four incident vertices are stored for each tetrahedron, plus four neighboring elements of this tetrahedron. Meanwhile, for each mesh vertex, one element incident to this vertex is stored. This data structure requires a small amount of memories, and the *shell-find* routine based on it only needs to traverse the elements locally. Given two ending vertices of an edge (denoted by v_1 and v_2, respectively), the *shell-find* routine is separated into two phases. In the first phase, one element that contains the input edge is searched by the following steps:
(1) Get the stored element incident to v_1 and push it into a stack;
(2) If the stack is empty, exit the routine and return NULL; otherwise, remove the top

element from the stack and go to Step 3;
(3) If the top element contains v_2, return the top element and exit the routine;
(4) Flag the top element as *visited*, and then visit its neighboring elements. For any unvisited neighbor, if it contains v_1 as well, push it into the stack;
(5) Go back to Step 2.

If no valid element is returned by the above procedure, the shell is empty; otherwise, starting from the returned element, the entire shell can be visited by using the neighboring indices of elements. In the worst scenarios, the first phase visits all elements surrounding v_1, and the number of such elements is close to 30 on average for real mesh examples. The second phase visits all elements surrounding the edge, and the number of such elements is about 5-7 on average[‡]. Therefore, the first phase dominates a general calling of the routine in terms of computing time. However, in many circumstances, one element surrounding an edge is stored somewhere before calling the shell-find routine. By using this element as an extra input, the timing-related performance of the shell-find routine can be improved remarkably by skipping over the first phase.

5.3 Validity and optimality conditions.

The shell transformation routine presented in Algorithm 3 needs to output an *optimal* covering mesh of a shell among all *valid* ones. Here, the definitions of validity and optimality depend on specific application purposes. For mesh quality improvement, three types of validation conditions are set for covering meshes as:

(1) *The basic condition*, which requires all elements have positive volumes.
(2) *The recursive condition*, which requires each shell transformation should create no supporting faces around any *ancestor edge* of the current supporting edge.
(3) *The application specific condition*, in the context of mesh improvement, which requires the quality of the output covering mesh of the shell should be higher than the quality of the input covering mesh.

It is possible that more than one *valid* covering mesh exists for a shell. The final output of a shell transformation calling is the covering mesh with the highest possible quality. See Section 5.1 for the definition of *mesh quality*.

5.4 The shell transformation based local reconnection scheme

If one edge or face of a low-quality element is removed, the element will be removed accordingly. Based on this concept, Algorithm 5 presents a local reconnection scheme that attempts to remove low-quality elements by removing the edges or faces of these elements.

All of low-quality elements are stored in a heap in an ascending order of the element quality. Firstly, Algorithm 4 is called on an edge of the first element of the heap. If the element is removed by Algorithm 4, Algorithm 4 succeeds; otherwise, Algorithm 4 is repeated on another edge of the element until all edges of the element are attempted. To protect the mesh boundary, the edges attempted for removal must be interior edges of the mesh. Next, if Algorithm 4 fails to remove the element, we attempt to remove the faces of this element individually. To protect the mesh boundary, the faces attempted for removal must be interior faces of the mesh. The 2-3 flip shown in Figure 1a is the simplest local scheme for face removal. However, a more effective alternative is *multi-face removal* (see Figure 1c).

[‡] It is worth noting that both numbers could vary case by case, depending on mesh topologies. Nevertheless, the meshes considered in this study are inputs for numerical simulations, where only a small percentage of elements are badly shaped. For different meshes of this type, it is observed that both numbers usually remain within the ranges we mentioned in the text. For instance, for the unimproved F16 and Bridge meshes to be presented in Section 7 for tests, the average numbers of elements surrounding interior mesh nodes are both 5.50. After mesh improvement, these two numbers are reduced to 5.21 and 5.20 respectively. The numbers of elements surrounding interior mesh edges are 26.05 and 26.21, respectively. After mesh improvement, these two numbers are reduced to 23.82 and 23.80 respectively.

Shewchuk **[10]** suggested an implementation of multi-face removal. In this study, we present an alternative solution based on the proposed shell transformation routine (i.e., Algorithm 3).

Given an interior face *f* for removal, the proposed algorithm takes the following steps as:

(1) Find two tetrahedra sharing face *f*, and denote the apexes of the tetrahedra opposite to *f* as *a* and *b*, respectively.

(2) Find all of the faces opposite to *a* and *b*. As shown in Figure 9a, these faces may form several connected components.

(3) Select a component that includes the face (or faces) intersected by *ab*, as shown in Figure 9b.

(4) Define the boundary of the selected component as a polygon. If a point (such as p_7 in Figure 9c) is contained in the interior of the polygon, remove one face (such as the face $p_1p_2p_7$ in Figure 9c) incident on this point from the component.

(5) Now we get a local mesh like the one shown in the left of Figure 1c. With this mesh as the input, *the shell transformation routine* (i.e., Algorithm 3) is called to search for a better covering mesh to fill in the shell region.

To avoid an infinite execution of the loop defined in Lines 2-16 of Algorithm 5, no matter the element for removal is removed or not, this element must be removed from the heap before the next iteration.

Algorithm 5. The combinational edge removal based on recursive shell transformation

```
localReconnection(M, l_max)
Inputs:
    the mesh to be improved, denoted M
    the maximally allowed recursive level, denoted l_max and the default value is 5
Variables:
    the heap that stores all of low-quality elements, T_bad
 1. Insert all of low-quality elements into T_bad in the ascending order of the element quality
 2. while T_bad is not empty
 3.    t: the first element of T_bad
 4.    If t has been removed from M
 5.       goto line 16
 6.    E = {e_1, e_2, …, e_n}: the set of edges of t qualified for removal (n <= 6)
 7.    for j = 1 to n
 8.       recursiveST(e_j, ∅, 0, l_max)
 9.       if t is removed
10.          goto line 16
11.    F = {f_1, f_2, …, f_m}: the set of faces of t qualified for removal (m <= 4)
12.    for j = 1 to m
13.       Remove f_j by a shell transformation based multi-face removal routine
14.       if t is removed
15.          break
16.    Remove t from T_bad
```

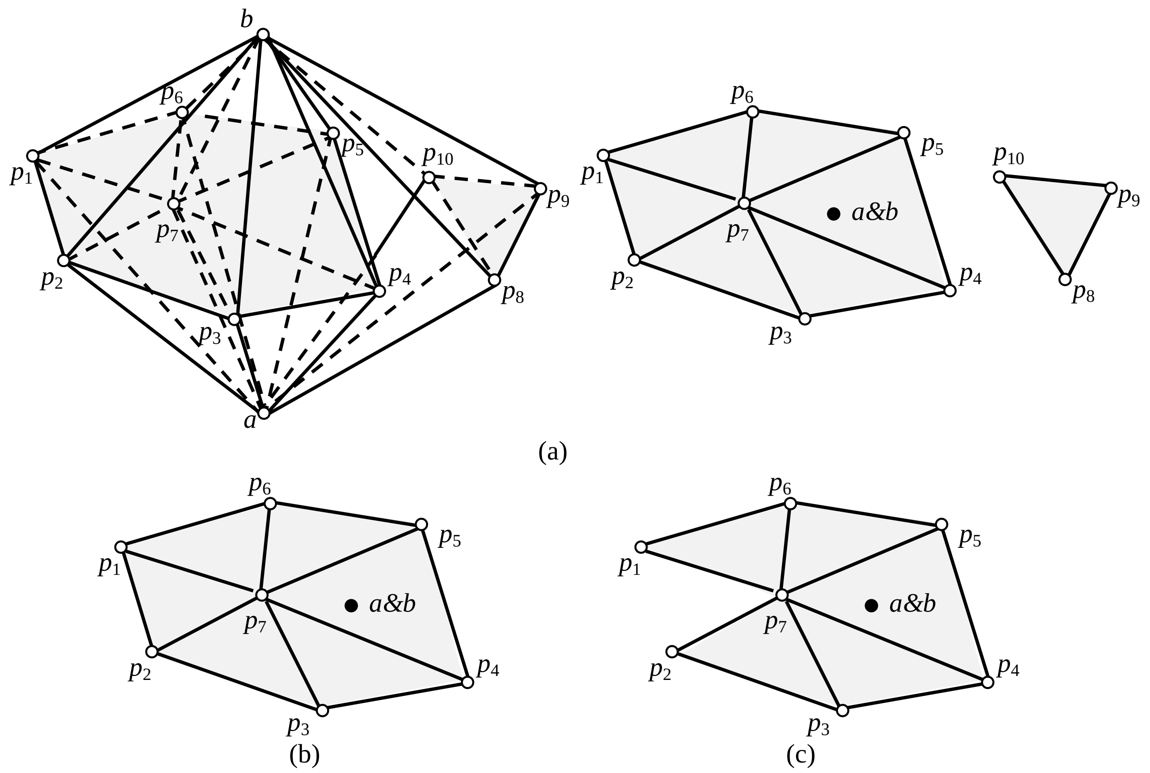


Figure 9. The procedure that prepares the inputs for the multi-face removal operation.

6. THE OVERALL MESH IMPROVEMENT ALGORITHM

The application goal of this study is to develop a cost-effective mesh improver. In this section, we first present some basic considerations that guide this development procedure. Then, we introduce the set of smoothing, point insertion and point suppression schemes incorporated in our mesh improver. Finally, the overall mesh improvement scheme is detailed.

6.1. The basic considerations

In this study, the *minimum sine* of dihedral angles is used as a default quality measure. The quality measure of a mesh is evaluated by a vector listing the quality of each tetrahedron contained by the mesh, in an order from the worst to the best. Since the worst tetrahedron in a mesh has far more influence than those average tetrahedra, the quality vectors of two meshes are compared *lexicographically* so that, for instance, an improvement in the second-worst tetrahedron improves the overall mesh quality even if the worst tetrahedron has not changed.

To ensure the heuristic algorithm never worsens the quality of a mesh, a *hill-climbing* method is adopted in all of the developed local schemes, which considers applying a local operation only if the quality of the changed mesh will be better than that of the original mesh. Local operations that do not improve the mesh quality are not applied. The method stops when no operation can achieve further improvement (i.e., the mesh is already locally optimal), or when a further optimization promises too little gain.

Meanwhile, since the focus is usually on the worst tetrahedron, only *bad elements* are treated to save the computing time, which refer to those elements whose minimum sine values are less than 0.5 in the following discussions, i.e., at least one dihedral angle of the element is either below 30° or above 150°.

The surface boundary of a volume mesh influences the mesh quality considerably. In the applications where the boundary can be changed to some extents, it is beneficial to extend the local schemes for mesh quality improvement from interior mesh entities to boundary entities.

However, in many applications, the improved mesh need to be consistent with a CAD model or matched face-to-face with another mesh. To limit the discussions, the presented algorithm regards the boundary configuration of the mesh as *untouchable*, i.e., vertices on the boundary cannot be smoothed, and the connectivity between them cannot be changed.

6.2. Smoothing

To achieve the cost-effectiveness, we combine an optimization-based algorithm **[20]** with the Laplacian smoothing to reposition each interior mesh point that is included by at least one bad element (referred to as a *bad point* hereafter):

(1) Perform Laplacian smoothing. If the improved *ball* (referring to all of elements incident on the point) contains no bad elements, the smoothing succeeds; otherwise, continue.

(2) Perform the optimization-based smoothing.

To save the smoothing time, a mesh point is flagged as *smoothed* after a successful smoothing, and this flag is flushed only if the ball of the point is changed. In each smoothing *cycle*, all of *non-smoothed bad* points are treated only once.

In each smoothing pass, the smoothing cycle is repeated until three indicators of the mesh quality are not improved further: (1) the quality of the worst tetrahedral (q_{worst}); (2) the number of bad elements (n_{bad}); and (3) the average quality of bad elements (q_{aver}).

6.3. Point insertion and point suppression

We adopt an *edge-splitting* based point insertion scheme. It attempts to insert a point at the middle of an interior edge and then to split those elements that meet at this edge; see Figure 10a. Finally, the new point is smoothed, and if the resulting mesh is better than the old one, the mesh will be changed; otherwise, the old mesh is restored.

Besides, our mesh improver relies on an *edge contraction* operation to remove *bad* points of the mesh. Figure 10b illustrates this operation using a 2D example. Each edge ended with the point to be suppressed is contracted to the other endpoint, and the resulting mesh configuration with the best quality is selected for further smoothing. To save computing time, only the point that replaces the contracted edge is smoothed. The point suppression operation fails if edge contraction (plus point smoothing) cannot produce a better mesh than the old one.

Since only bad elements are targeted, those edges included by bad elements are attempted only once in each point suppression or insertion pass.

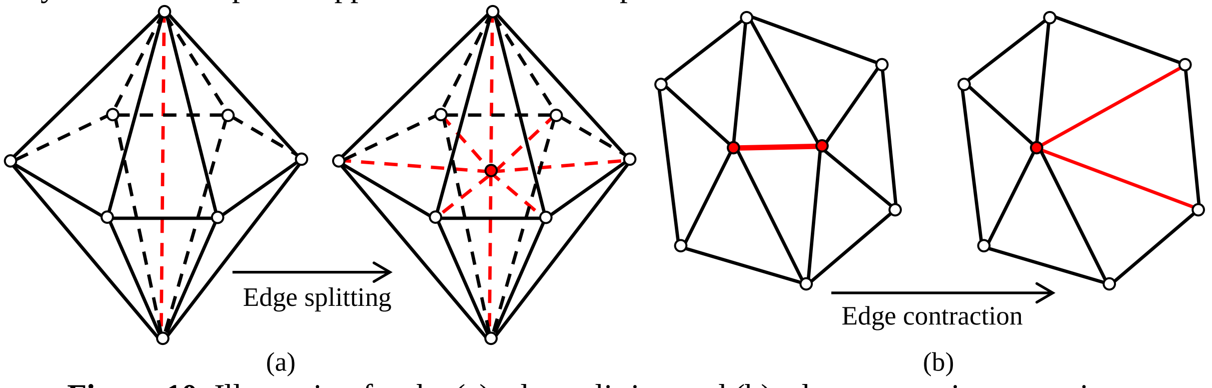


Figure 10. Illustration for the (a) edge splitting and (b) edge contraction operations.

6.4. The mesh improvement schedule

Algorithm 6 presents the proposed mesh improvement schedule, which combines different local schemes to improve the mesh quality. This schedule begins with a smoothing pass, and then executes the main loop of mesh improvement. In the main loop, a smoothing pass is followed after the pass of each type of topological transformations to improve the mesh quality further. The main loop is ended when three subsequent combinational passes fail to make sufficient progress or the number of iteration steps exceeds a predefined threshold (in the present study, the default value of this threshold is 30). We gauge progress using three quality indicators mentioned in Section 5.4, i.e., q_{worst}, n_{bad} and q_{aver}.

Algorithm 6. The proposed mesh improvement schedule

```
improveAMesh(M)
Input:
    M, the mesh to be improved
Variables:
    $q_{worst}$, $q'_{worst}$, the quality of the worst tetrahedral
    $n_{bad}$, $n'_{bad}$, the number of bad elements
    $q_{aver}$, $q'_{aver}$, the average quality of bad elements
  1.  failed = 0; itcount = 0
  2.  Smooth M
  3.  Query the mesh quality and store the indicators in $q_{worst}$, $n_{bad}$ and $q_{aver}$, respectively
  4.  while failed < 3 && ++itcount <= 30
  5.      localReconnection(M, 5)
  6.      Smooth M
  7.      Improve M by the point suppression scheme
  8.      Smooth M
  9.      Improve M by the point insertion scheme
  10.     Smooth M
  11.     Query the mesh quality and store the indicators in $q'_{worst}$, $n'_{bad}$ and $q'_{aver}$, respectively
  12.     if ($q'_{worst} < q_{worst}$ || $n'_{bad} < n_{bad}$ || $q'_{aver} < q_{aver}$) failed = failed + 1
  13.     else failed = 0
  14.     $q_{worst} = q'_{worst}$; $n_{bad} = n'_{bad}$; $q_{aver} = q'_{aver}$
```

7. RESULTS

The numerical tests are conducted on a PC workstation (CPU: 3.5GHz, Memory: 24GB). Results obtained from the developed mesh improver are compared with those obtained by Grummp (Version 0.3.4) and Stellar (Version 1.0), respectively. To our knowledge, Grummp **[20, 32]** and Stellar **[12, 13]** are among the best open-source improvers for tetrahedral meshes. Although many common features exist between two codes, their differences are also evident due to different start points of their development. The goal of Grummp is to improve the worst tetrahedra cost-effectively, while the goal of Stellar is to improve the worst tetrahedra aggressively with speed as a secondary consideration.

In default, Grummp code takes four steps to improve a given mesh:

(1) Perform three passes of local reconnections for all elements.

(2) Perform two smoothing passes for elements containing angles below θ degrees or above 180-θ degrees. Here, θ is a threshold initially set as 25° and then adaptively reduced after each pass of smoothing operations.

(3) Repair a small fraction of the worst tetrahedra by a full range of swapping techniques.

(4) Repeat Step 2.

The default schedule of Stellar code begins with one smoothing pass, one local reconnection pass and one edge contraction pass for all elements, and then combines these local schemes and a point insertion scheme in a loop to improve the quality of a mesh iteratively. Inside this loop, the smoothing and local reconnection routines target at all elements, but the most passes of edge contraction and point insertion routines target at elements with angles below 40° or above 140° except for a so-called *desperation pass*, which targets at the worst 3.5% of tetrahedra.

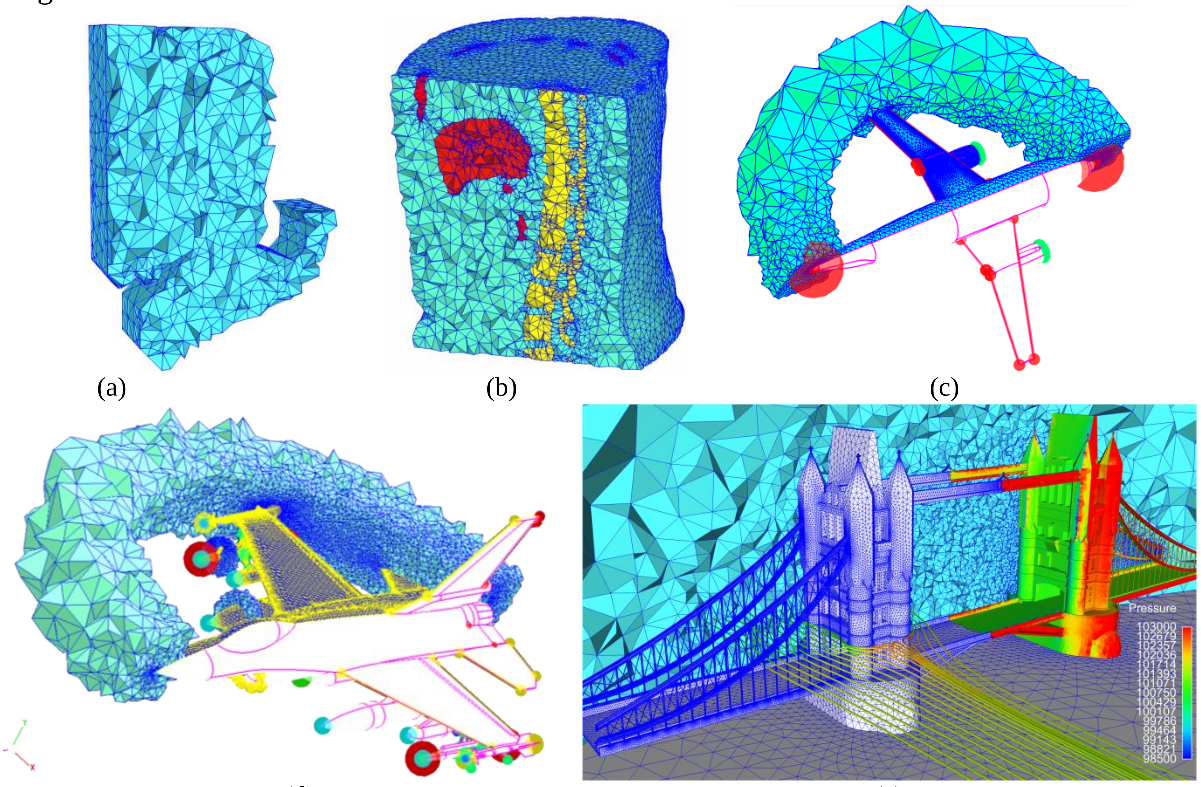


Figure 11. The selected meshes: (a) Tire; (b) Patient-organs02; (c) The F6 aircraft; (d) The F16 aircraft; (e) The London Tower bridge. Surface meshes and grid sources for mesh sizing control are displayed in both graph (c) and graph (d), and the surface mesh and inviscid flow simulation results induced by a crosswind are displayed in graph (e), respectively.

As shown in Figure 11, five meshes are selected. The first two meshes are accessible from the internet: the initial mesh of *tire* was ever analysed in literatures **[20]** and **[32]**, and included in the package of Grummp **[32]** and Stellar **[13]**, and the initial mesh of *patient-organs02* is obtained from the AIM@SHAPE repository **[33]**. The last three meshes are generated by our in-house codes following the same schedule as:

(1) Input a geometry model.
(2) Triangulate the surface by an advancing front technique.
(3) Tetrahedralize the volume by a Delaunay mesher **[4]**.

The original geometry models of the last three examples are all accessible from the internet. The London Tower Bridge model (referred to as *bridge* hereafter) and the DLR-F6 wing-body-nacelle-pylon aircraft model (referred to as *F6* hereafter) are the selected test case geometries for the meshing contest session of the 23rd International Meshing Roundtable (IMR) and the 2nd AIAA CFD Drag Prediction Workshop, respectively. The F16 aircraft model (referred to as *F16* hereafter) is obtained from GrabCAD **[34]**.

Table 1 lists the initial mesh size statistics and the mesh quality data of those selected examples, where $\theta_{\min}$ and $\theta_{\max}$ refer to the minimum and maximum dihedral angles, and λ refers to the percentage of bad dihedral angles, i.e., angles within the range of [0, 30°] or [150°, 180°], respectively. Meanwhile, λ_i (i=1-5) is used to evaluate the distributions of bad angles, which refers to the percentage of dihedral angles within the range of [6(i-1), 6i) or (180-6i, 180-6(i-1)] degrees. For instance, λ_2 refers to the percentage of dihedral angles within the range of 6° to 12° or 168° to 174°.

Table 1. The initial mesh size statistics and mesh quality data.

Examples	#tetra.	#points	$\theta_{\min}$ (°)	$\theta_{\max}$ (°)	% of bad angles (λ)	Distribution of bad angles (%)				
						λ_1	λ_2	λ_3	λ_4	λ_5
Tire	11,098	2,570	0.66	178.88	4.58	0.12	0.46	0.77	1.28	1.96
Patient-organs02	280,911	51,124	2.89	175.23	7.65	0.0072	0.21	1.20	2.45	3.78
F6	1,023,532	172,664	2.6e-13	≈180	6.65	0.23	0.68	1.19	1.82	2.73
F16	18,065,336	2,906,056	2.6e-14	≈180	6.63	0.23	0.67	1.18	1.82	2.74
Bridge	37,772,656	6,205,571	2.1e-13	≈180	6.72	0.23	0.68	1.19	1.85	2.77

In the first test, the performance data of different local reconnection schemes are compared with each other. Stellar code executes edge removal and multi-face removal routines repeatedly to improve mesh topology, while Grummp code executes 2-3 flips and edge removal routines repeatedly. We improve the five initial meshes by performing one pass of the three local reconnection schemes, respectively. Instead of improving all elements, only *bad* elements are treated in this test. Table 2 presents the mesh quality and the computing time data comparison.

For *patient-organs02*, *F6* and *F16* cases, our scheme not only achieves the lowest percentage of bad angles (λ), but also narrows the ranges of dihedral angles to the largest extent. For *bridge* case, our scheme also achieves the lowest λ; however, all of the three local reconnection schemes fail to improve both the smallest and the largest angles to an acceptable level, although the values achieved by our scheme are slightly better. For *tire* case, Grummp code improves $\theta_{\min}$ and $\theta_{\max}$ at the same level as our scheme. Meanwhile, Grummp code achieves a slightly better value of λ than our scheme, while our scheme achieves smaller values of λ_1 and λ_2. We believe that, for this mesh, more angles between 12° and 30° (or between 150° and 168°) are generated when our scheme attempts to remove small angles between 0° and 12° or large angles between 168° and 180°, because the cost of improving the worst angle is possibly increased considerably due to the generation of more undesirable small/large angles.

In this test, Stellar code achieves the best performance with respect to the computational time, while Grummp code performs rather well for small meshes but very poor for big meshes. For our scheme, one pass of the proposed local reconnection scheme consumes more time than its counterpart in Stellar code, because of its recursive nature. Nevertheless, since the proposed scheme produces a much better mesh, this marginally more time consumption is acceptable.

Table 2. The mesh quality and computational time data for different local reconnection schemes.

Examples		#tetra.	$\theta_{\min}$ (°)	$\theta_{\max}$ (°)	λ	Distribution of bad angles (%)					Time (s)
						λ_1	λ_2	λ_3	λ_4	λ_5	
Tire	Grummp	11,019	3.36	172.38	4.21	0.069	0.30	0.57	1.18	2.09	0.09
	Stellar	10,936	3.00	172.38	4.23	0.056	0.31	0.59	1.23	2.05	0.07
	Present	10,906	3.36	172.38	4.42	0.047	0.27	0.61	1.30	2.19	0.21
Patient-organs02	Grummp	265,086	5.68	165.38	3.46	6.3e-5	3.3e-3	0.12	0.96	2.37	2.6
	Stellar	261,485	6.55	167.70	2.98	0	2.4e-3	0.089	0.79	2.09	2.6
	Present	259,959	11.21	162.59	2.81	0	6.4e-5	0.034	0.67	2.10	5.9
F6	Grummp	955,584	0.78	178.4	0.75	3.0e-4	2.0e-3	0.016	0.10	0.63	9.8
	Stellar	951,623	0.73	178.7	0.58	3.7e-4	4.3e-3	0.022	0.094	0.46	4.6
	Present	946,534	3.69	174.85	0.31	7.0e-5	4.8e-4	2.5e-3	0.018	0.29	5.7
F16	Grummp	16,854,123	2.4e-4	≈180	0.74	3.9e-4	2.3e-3	0.018	0.11	0.61	258.0
	Stellar	16,791,528	1.3e-4	≈180	0.57	9.5e-4	4.7e-3	0.024	0.097	0.45	72.6
	Present	16,682,773	3.80	174.20	0.31	1.9e-5	4.1e-4	4.1e-3	0.025	0.28	85.3
Bridge	Grummp	35,233,789	3.0e-5	≈180	0.81	2.5e-4	2.0e-3	0.017	0.11	0.67	542
	Stellar	35,081,252	9.2e-5	≈180	0.62	4.0e-4	3.8e-4	0.022	0.10	0.50	160.8
	Present	34,853,521	8.6e-4	≈180	0.36	1.5e-5	1.6e-4	1.9e-3	0.025	0.33	192.2

In the second test, we compare the default schedules of Grummp code, Stellar code and the proposed algorithm (i.e., Algorithm 6). In this test, the option that prohibits the change on the mesh surface is enabled for both Grummp and Stellar codes. In addition, because the first pass of edge contraction in Stellar code can coarsen the input mesh dramatically, this pass is thus disabled in this test.

Table 3 presents the mesh quality and computational time data from the second test. In all of the cases, Grummp code outputs the worst quality meshes. For *F16* and *bridge* cases, the meshes output by Grummp code contain extremely small and/or large angles, while Stellar code and our algorithm can improve them to an acceptable level for further numerical simulations. We believe that the following facts might account for the relatively poor performance of Grummp code. Firstly, Grummp code does not incorporate any point suppression and point insertion schemes. In practice, these two schemes are useful for eliminating extreme small and/or large angles of a mesh. Secondly, Grummp code only executes a fixed number of passes of swapping and smoothing operations. In both Stellar code and our algorithm, the adopted scheduling strategies that combine local mesh improvement schemes are far more aggressive.

Table 3. The mesh quality and computational time data of the default schedules of Grummp, Stellar and our improved method.

Examples		#tetra.	#points	$\theta_{\min}$ (°)	$\theta_{\max}$ (°)	λ	Distribution of bad angles (%)					Time (s)
							λ_1	λ_2	λ_3	λ_4	λ_5	
Tire	Grummp	11,039	2,570	13.67	158.55	1.7	0	0	0.030	0.21	1.47	0.5
	Stellar	10,973	2654	23.4	148.1	0.15	0	0	0	0.017	0.13	106
	Present	11,840	2,751	20.67	157.45	0.26	0	0	0	0.018	0.24	1.0
Patient -organs02	Grummp	264,954	51,124	8.93	160.40	2.91	0	1.3e-4	1.6e-3	0.06	2.85	21
	Stellar	227,775	46,237	31.7	141.58	0	0	0	0	0	0	3,220
	Present	266,631	52,392	20.52	149.93	8.3e-3	0	0	0	3.8e-4	7.9e-3	16
F6	Grummp	955,512	172,664	0.78	178.4	0.65	1.7e-4	3.0e-4	1.1e-3	3.5e-3	0.65	43
	Stellar	918,434	171,912	18.2	158.7	5.9e-3	0	0	0	4.7e-4	5.4e-3	1,193
	Present	935,608	172,167	10.64	159.79	0.017	0	5.3e-5	3.2e-4	1.4e-3	0.015	26
F16	Grummp	16,854,105	2,906,056	2.4e-4	179.99	0.63	1.5e-4	3.2e-4	2.6e-3	0.010	0.62	884
	Stellar	16,360,678	2,910,076	3.8	174.2	0.027	6.1e-6	3.1e-6	1.1e-4	9.1e-4	0.026	9,882
	Present	16,666,517	2,923,840	3.8	174.2	0.013	6.0e-6	1.3e-5	7.7e-5	1.2e-3	0.012	522
Bridge	Grummp	35,233,695	6,205,571	3.0e-5	179.99	0.71	1.1e-4	2.0e-4	8.6e-4	6.8e-3	0.70	1,521
	Stellar	34,188,981	6,211,120	7.5	171.5	0.13	0	1.0e-4	4.2e-4	9.5e-4	0.13	21,779
	Present	33,698,234	6,091,180	5.79	172.55	0.13	9.9e-7	7.0e-5	4.0e-4	1.9e-3	0.13	1,412

In the second test, Stellar code outputs the best quality meshes in most occasions, and it not only narrows the range of dihedral angles at most, but also reduces the percentage of bad angles as well. For instance, for *patient-organs02*, Stellar code improves the smallest and the largest angles to be 31.7° and 141.6°, respectively. In other words, the improved mesh does not contain any bad angles. The mesh improved by our algorithm is slightly worse, which contains 132 angles below 30° (of which 6 angles below 24°), but no angles above 150°. For *F16* and *bridge* cases, the quality levels of the improved meshes by Stellar code and our algorithm are very close.

With respect to the computational time performance, the proposed algorithm performs the best in most cases, apart from the improvement of the mesh *tire*. For this smallest mesh, Grummp code performs the best in terms of the timing performance. Here, we define a *velocity* index to evaluate the timing performance as:

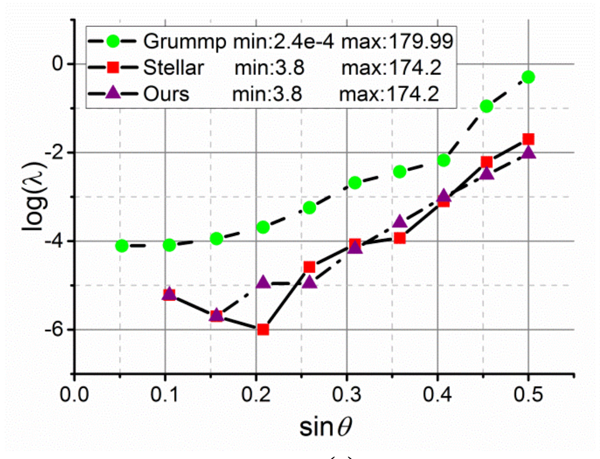
v = the number of elements contained in the input mesh / the total computing time.

For those three inputs produced by the Delaunay mesher, i.e., *F6*, *F16* and *bridge*, Stellar code runs at a speed of 45.9, 18.9 and 15.4 times slower than that of the proposed algorithm, respectively. It is worth noting that the adopted Delaunay mesher runs very fast. For instance, the generation of the initial mesh of *bridge* consumes only 174 seconds, while the proposed algorithm can improve it to an acceptable mesh quality level for simulations in about 1,412 seconds. However, if replacing the proposed algorithm by Stellar code, it will take about 6 hours to achieve only marginal improvement (compared with our results) in terms of the mesh quality. In this respect, the proposed algorithm is undoubtedly a more *cost-effective* choice than the current default schedule of Stellar code.

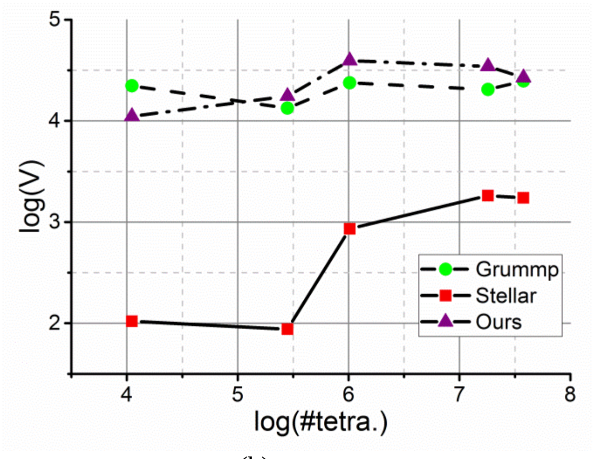
To demonstrate the performance difference between three mesh improvers more clearly, Figure 12a compares the distributions of bad angles of three F16 meshes produced by Grummp code, Stellar code and our mesh improver. By default, each curve contains 10 data points, and the λ value of the *i*th point refers to the percentage of bad dihedral angles within the range of [3(*i*-1), 3*i*) or (180-3*i*, 180-3(*i*-1)] degrees (*i*=1-10). Nevertheless, because the meshes produced by Stellar code and our mesh improver contains no angles below 3° or above 177°, their corresponding curves contains no data points referring to angles within this range. Besides, Figure 12b compares the timing performance of three mesh improvers for five test meshes of various sizes, evaluated by the velocity indices of these improvers mentioned previously.

The λ value of each data point of the curve for Grummp code is found larger than its counterparts from Stellar code and our mesh improver by nearly one or two orders of magnitude, while the λ values of data points of the curves for Stellar code and our mesh improver are comparable in general. However, the velocity indices of Stellar code are lower than their counterparts of Grummp code and our mesh improver by one or two orders of magnitude, while the velocity indices of Grummp code and our improver is at the same order. From the above analysis, we can conclude that our mesh improver presented in this study can achieve an overall better balanced performance between the mesh quality and computational time than other two state-of-the-art algorithms, and it is therefore more suitable for the quality improvement tasks involving large-scale meshes.

It needs to be pointed out that the initial mesh of *patient-organs02* actually contains interior constraints. However, because the current versions of Grummp and Stellar codes provide no options to input a mesh with interior constraints, all of the tests presented above choose not to respect interior constraints. In fact, the proposed algorithm can respect interior constraints very well. To demonstrate this, Figure 13 compares the meshes improved by the proposed algorithm with or without interior constraints. Not surprisingly, the quality of the improved mesh that respects interior constraints is slightly worse.



(a)



(b)

Figure 12. A comparison of Grummp, Stellar and our improver in terms of mesh quality and timing performance. (a) The distributions of bad angles of the F16 meshes produced by three improvers. (b) The velocity indices of three improvers for five test meshes.

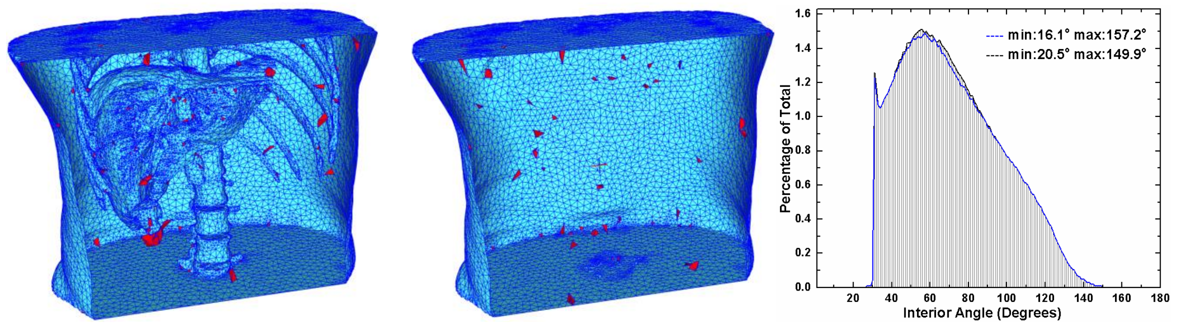


Figure 13. A comparison of the improved meshes of *patient-organs02* by the proposed algorithm with interior constraints respected or not. In graphs (a) and (b), the blue triangles are boundary triangles, and the red tetrahedra are elements containing dihedral angles below 30° or above 150°. Graph (c) compares the distributions of dihedral angles of both meshes.

Finally, the applicability of the developed mesh improver for real aerodynamics simulations is demonstrated by a store separation simulation of a fully-loaded F16 aircraft. In this test, four stores are separated from the aircraft to verify the robustness of our in-house CFD system for complex flow simulations **[35]**. The main loop of this simulation includes four main steps:

(1) Compute the unsteady flow by a finite volume solver.
(2) Compute aerodynamic forces and moments based on flow simulation results, with which as inputs, the positions of moving bodies in the next time step are determined using the six degrees-of-freedom equations of motion.
(3) Move the mesh points to adapt the movement of mesh boundaries.
(4) If mesh movement yields elements with unacceptable quality, the holes are formed by deleting these elements. Next, a new mesh is formed by merging undeleted elements and new elements filled in the holes. Finally, the solution is reconstructed by interpolation.

The initial volume mesh is composed of about 3.69 million tetrahedral elements (see Figure 14a). Figures 14b and 14c compare the meshes before and after local remeshing at t_s = 0.147s (t_s refers to a physical time of separation). Figures 14d presents the mesh at t_s = 0.3s, instantly after another local remeshing step is accomplished. Because the simulation involves very complicated boundary movements, the proposed remeshing algorithm is employed very frequently. Considering the simulation process until t_s = 0.5s, the remeshing algorithm is employed for a total of 44 times. On average, each local remeshing step needs to generate and improve a local mesh size composed of about 350K elements. Stellar code is obviously inappropriate for this kind of application because of its huge time consumption. Before this study, Grummp code was ever employed for a local mesh improvement. It was observed that Grummp code occasionally failed to provide a qualified mesh for simulations. One possible reason could be that mesh faces are largely stretched during the mesh deformation process and some of them may even appear on the boundaries of the holes to be remeshed. Grummp code sometimes failed to remove those low-quality elements attaching to these stretched faces. After replacing Grummp code with the present mesh improver, no failing case has been reported as far as this simulation is concerned.

Note that only the steps of the CFD solution and mesh deformation were executed in parallel on 32 computer cores in this test, while other steps, including local remeshing, are executed sequentially. Not surprisingly, the CFD solution step is most time-consuming, which used 91.9% of the total computing time. The mesh deformation step only used 3.9% of the total computing time because a simple spring-analogy approach was adopted **[35]**. Although local remeshing calling is executed sequentially, its total time cost is very low (using only 2.9% of the total computing time).

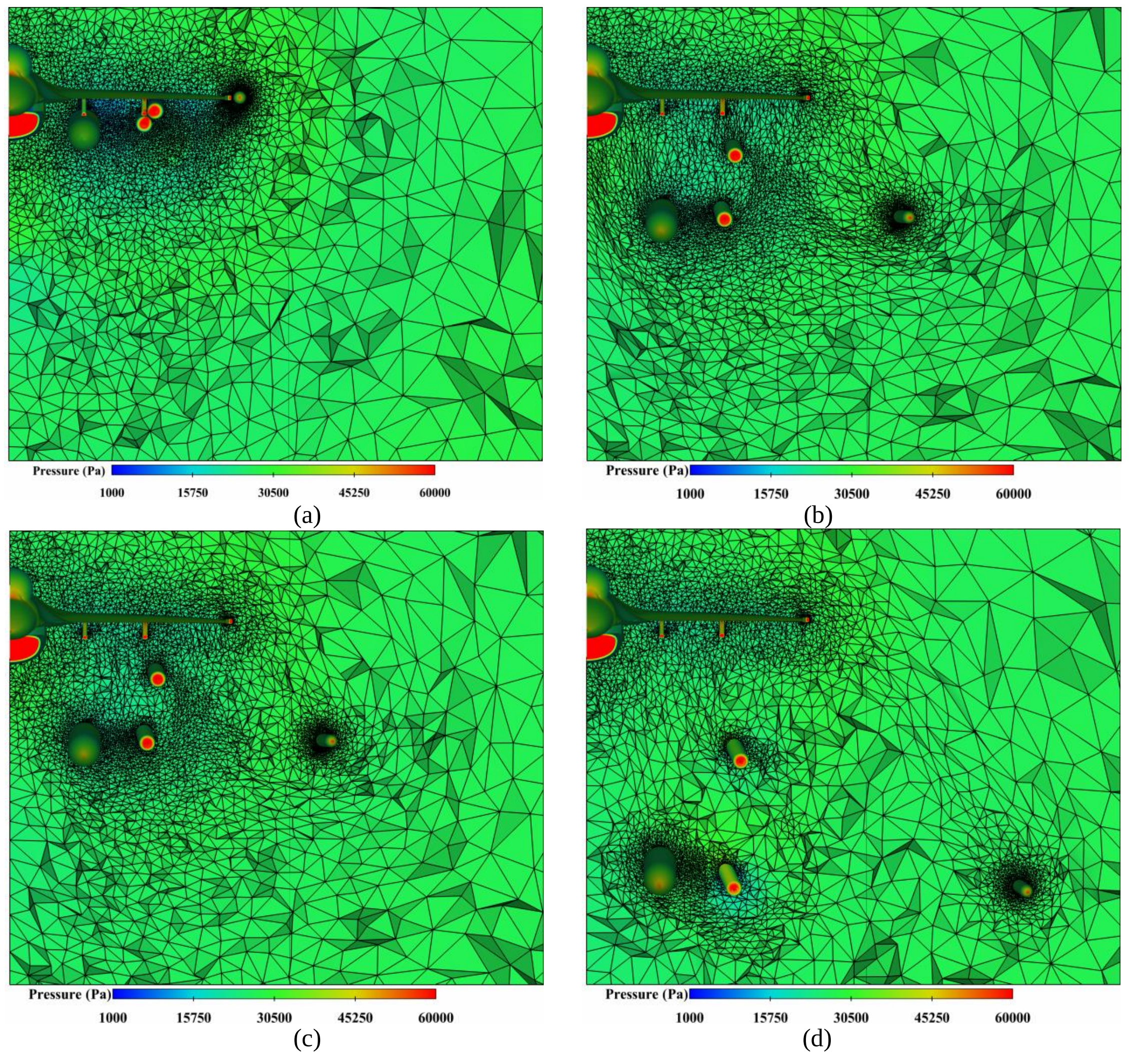


Figure 14 Cut views of the volume meshes for the store separation problem of a full-loaded F16 aircraft at different physical time of separation (t_s): (a) t_s= 0s; (b) t_s = 0.147s, before local remeshing; (c) t_s = 0.147s, after local remeshing; (d) t_s= 0.3s, after local remeshing.

8. CONCLUSIONS

A new flip named shell transformation is proposed for mesh quality improvement. Its single calling could be considered as an enhanced version of the edge removal transformation, while its recursive scheme acts like “composite edge removal transformations”. In practice, this recursive scheme provides an elaborate pattern to combine multiple flips and perform these flips on hundreds of elements for a single goal, for instance, removing a low-quality element by removing one of its boundary edges. Accordingly, the possibility to achieve such a goal by the proposed recursive scheme is much higher than those based on performing single flips individually.

Furthermore, a new mesh improvement algorithm is developed by combining the proposed local reconnection scheme with smoothing and other topological transformation schemes. Numerical experiments have revealed that the proposed algorithm is capable of balancing the requirements for a high-quality mesh and a low computational time costs spent on the mesh improvement for large-scale engineering flow problems.

Acknowledgements: The authors appreciate the joint support for this project by the National Natural Science

Foundation of China (Grant No. 11172267, 11432013, 10872182) and Zhejiang Provincial Natural Science Foundation (Grant No. LR16F020002 and Y1110038). The first author acknowledges the joint support from Zhejiang University and China Scholarship Council and the host of Professor Oubay Hassan and Professor Kenneth Morgan for his recent research visit to Swansea University, UK.

REFERENCES

[1] Weatherill NP, Hassan O. Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints. *Int. J. Numer. Methods Eng.* 1994; **37**:2005-2039.

[2] George PL, Borouchaki H, Saltel E. 'Ultimate' robustness in meshing an arbitrary polyhedron. *Int. J. Numer. Methods Eng.* 2003; **58**:1061-1089.

[3] Du Q, Wang D. Constrained boundary recovery for three dimensional Delaunay triangulations. *Int. J. Numer. Methods Eng.* 2004; **61**:1471-1500.

[4] Chen J, Zhao D, Huang Z, Zheng Y, Gao S. Three-dimensional constrained boundary recovery with an enhanced Steiner point suppression procedure. *Comput. Struct.* 2011; **89**:455-466.

[5] Si H. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Software* 2015; **41**:11:1-11:36.

[6] Joe B. Construction of three-dimensional improved-quality triangulations using local transformations. *SIAM J. Sci. Comput.* 1995; **16**:1292-1307.

[7] De L'Isle EB, George PL. Optimization of tetrahedral meshes. *IMA Vol. Math. Appl.* 1995; **75**:97-128.

[8] de Cougny HL, Shephard MS. Refinement, derefinement, and optimization of tetrahedral geometric triangulations in three dimensions.1995. Unpublished manuscript.

[9] Misztal M, Bærentzen J, Anton F, Erleben K. Tetrahedral mesh improvement using multi-face retriangulation. *Proceedings of the 18th International Meshing Roundtable*, Salt Lake City, UT, USA, 2009; 539-555.

[10] Shewchuk JR. Two discrete optimization algorithms for the topological improvement of tetrahedral meshes. 2002. Unpublished manuscript. April-15-2016. URL: https://www.cs.berkeley.edu/~jrs/papers/edge.pdf.

[11] Klingner BM, Shewchuk JR. Aggressive tetrahedral mesh improvement. *Proceedings of the 16th International Meshing Roundtable*, Seattle, WA, USA, 2007; 3-23.

[12] Klingner BM, Shewchuk JR. Stellar: a tetrahedral mesh improvement program. Jan-21-2016. URL: http://www.cs.berkeley.edu/~jrs/stellar.

[13] Klincsek GT. Minimal triangulations of polygonal domains. *Ann. of Discrete Math.* 1980; **9**:121–123,

[14] Lo SH. 3D Delaunay triangulation of 1 billion points on a PC. *Finite Elem. Anal. Des.* 2015. **102-103**: 65-73.

[15] Chen J, Zhao D, Huang Z, Zheng Y, Wang D. Improvements in the reliability and element quality of parallel tetrahedral mesh generation. *Int. J. Numer. Methods Eng.* 2012; **92**:671-693.

[16] Zhao D, Chen J, Zheng Y, Huang Z, Zheng J. Fine-grained parallel algorithm for unstructured surface mesh generation. *Comput. Struct.* 2015; **154**:177-191.

[17] Löhner R. Recent advances in parallel advancing front grid generation. *Arch. Computat. Methods Eng.* 2014; **21**:127-140

[18] Joe B. Construction of three-dimensional Delaunay triangulations using local transformations. *Comput. Aided Geom. Des.* 1991; **8**:123-142.

[19] Compère G, Remacle JF, Jansson J, Hoffman J. A mesh adaptation framework for dealing with large deforming meshes. *Int. J. Numer. Methods Eng.* 2010; **82**:843-867.

[20] Freitag LA, Ollivier-Gooch C. Tetrahedral mesh improvement using swapping and smoothing. *Int. J. Numer. Methods Eng.* 1997; **40**:3979-4002.

[21] Liu J, Chen B, Sun S. Small polyhedron reconnection for mesh improvement and its implementation based on advancing front technique. *Int. J. Numer. Methods Eng.* 2009; **79**:1004 -1018.

[22] Field DA. Laplacian smoothing and Delaunay triangulations. *Commun. Appl. Numer. Methods* 1988; **4**:709–712.

[23] Freitag LA, Knupp PM. Tetrahedral mesh improvement via optimization of the element condition number. *Int. J. Numer. Methods Eng.* 2002; **53**:1377-1391.

[24] Jiao X, Wang D, Zha H. Simple and effective variational optimization of surface and volume triangulations. *Eng. Comput.* 2011; **27**:81-94.

[25] Leng J, Zhang Y, Xu G. A novel geometric flow approach for quality improvement of multi-component tetrahedral meshes. *Comput.-Aided Des.* 2013; **45**:1182-1197.

[26] Vartziotis D, Wipper J. Fast smoothing of mixed volume meshes based on the effective geometric element transformation method. *Comput. Methods Appl. Mech. Eng.* 2012; **201-204**:65-81.

[27] Vartziotis D, Wipper J, Papadrakakis M. Improving mesh quality and finite element solution accuracy by

GETMe smoothing in solving the Poisson equation. *Finite Elem. Anal. Des.* 2013. **66**: 36-52.

[28] Freitag DL, Knupp P, Munson T, Shontz S. A comparison of two optimization methods for mesh quality improvement. *Eng. Comput.* 2006; **22**:61-74.

[29] Cheng S-W, Dey TK, Shewchuk JR. *Delaunay Mesh Generation*. CRC Press, Boca Raton, Florida, December 2012.

[30] Bedregal C, Rivara MC. Longest-edge algorithms for size-optimal refinement of triangulations. *Comput.-Aided Des.* 2014; **46**:246–251.

[31] Ruppert J, Seidel R. On the difficulty of triangulating three-dimensional non-convex polyhedra. *Discrete Comput. Geom.* 1992; **7**:227-254.

[32] Ollivier-Gooch C. GRUMMP. April-15-2016. URL: http://tetra.mech.ubc.ca/GRUMMP.

[33] Model name: IRCADb patient-organs02 (model ID: 1477-IRCADb_patient-organs02). April-15-2016. URL: http://visionair.ge.imati.cnr.it:8080/ontologies/shapes/view.jsp?id=1477-IRCADb_patient-organs02.

[34] Guimarães GF. F-16 Fighting Falcon. April-15-2016. URL: https://grabcad.com/library/f-16-fighting-falcon.

[35] Zheng J, Chen J, Zheng Y, Yao Y, Li S, Xiao Z. An improved local remeshing algorithm for moving boundary simulations. *Eng. Appl. Comp. Fluid* 2016. **10**: 405-428.