Received 30 March 2024, accepted 10 April 2024, date of publication 12 April 2024, date of current version 29 April 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3388017

RESEARCH ARTICLE

An Efficient Slicing Algorithm for Additive Manufacturing in Arbitrary Directions Based on Adjacent Triangles

HAN LIU[1] **AND FEI XING**[1,2]
[1]School of Mechanical Engineering, Shenyang University of Technology, Shenyang 110870, China
[2]Nanjing Zhongke Raycham Laser Technology Company Ltd., Nanjing 210038, China

Corresponding author: Fei Xing (xingfei@raycham.com)

This work was supported by the National Key Research and Development Program of China under Grant 2022YFB4602200.

ABSTRACT Additive manufacturing is capable of forming part solids from a digital model of the workpiece, which often needs to be sliced due to the nature of the process of stacking materials layer by layer. Previous algorithms require more time to order the intersection points during slicing and only consider a fixed axis vector as the slicing direction. With the increasing complexity of the workpiece model, it is necessary to improve the efficiency of the slicing algorithm, and the given slicing direction should be flexible and convenient. Therefore, this paper proposes an efficient algorithm for slicing in any direction using the shared edges of adjacent triangles. Firstly, the model can be efficiently sliced in any direction by custom vectors, which is convenient for process validation; secondly, the shared edges of neighboring triangular facets intersecting the plane need to be calculated only once to reduce redundant calculations; finally, the ordered intersections of the constructed inner and outer contours are directly outputted without any additional sorting to reduce the running time. The experimental results show that compared with other algorithms, the proposed algorithm can improve the execution efficiency and eliminate the constraint that the slicing direction is fixed on the Z-axis, which is of application value and reference significance for additive.

INDEX TERMS Additive manufacturing, slicing algorithm, adjacency topology, calculation optimization.

I. INTRODUCTION

Additive manufacturing uses bottom-up stacking of materials to print workpieces, which is more capable of manufacturing geometrically complex parts than subtractive or equal material manufacturing [1]. Additive manufacturing converts a workpiece from a 3D model to a solid one, requiring process planning that includes positioning and orienting the workpiece model in the device space, meshing the workpiece's geometric model, and planning the machining path for each and between layers. Due to the characteristics of layered processing in additive manufacturing, slicing is a more critical step [2]. In additive manufacturing, there are many formats of 3D model files for parts, such as CAD, 3MF, OFF, OBJ, and STL formats. Among them [3], the STL file format is more widely used. After the slicing algorithm reads the coordinates of vertices and normal vectors of all the triangular facets from the STL file, in the first step, the intersections of the slicing planes and the triangular facets (triangle-plane intersections) are calculated through this crucial information and the set layer thickness. In the second step, the boundary of each layer's print area is constructed according to the desired intersections. These two steps are the standard processes for the previous algorithm to achieve slicing [4].

As the number of triangular facets required to fit the workpiece model increases, the amount of computation and execution time increases and even more memory is required [5]. Slicing algorithms usually need to go through two steps of intersection calculation and contour construction to get results [6]. Most algorithms will be optimized for these two steps respectively, but there are three problems in this way. First, they sort the triangles by the smallest vertex in the intersection calculation step, which would lose the topology of the triangle mesh, so it would take more time to sort the

The associate editor coordinating the review of this manuscript and approving it for publication was Walter Didimo.

intersections in the contour construction phase. Second, even if they use a less time complex data structure to store the intersection points in the contour building step, it still makes the running time longer. Third, these optimization processes limit the slice direction to the Z-axis, which brings inconvenience to the verification of the process. Therefore, in order to improve the execution efficiency of the algorithm and provide more choices for the process, this paper proposes an arbitrary direction slicing algorithm based on adjacent triangular facets. The rest of the paper is organized as follows: Section II reviews other slicing algorithms, Section III describes our proposed algorithm in detail, Section IV compares and analyzes the proposed algorithm with other algorithms to reflect the improvement effect, and Section V draws conclusions and summarizes.

II. RELATED WORKS

Previous slicing algorithms improve the algorithm by optimizing calculation, improving part quality, and reducing build time. This paper only focuses on reviewing related work that optimizes the calculation to improve the execution efficiency of the algorithm. Kirschman and Jara-Almonte [7] proposed the most straightforward slicing algorithm, in which all triangular facets and each layer of slicing planes intersect. Because the relative positions of most of the triangles and the current layer plane determine that they cannot cross, Tata et al. [8] adopted group sorting to reduce unnecessary traversal of the triangular facets in the subsequent process, thus reducing time and avoiding waste. Tian et al. [9] adopted a binary search method when arranging and grouping triangular facets in advance, thus reducing the number of judgments on the position relationship between triangular facets and the slicing planes. Although the slicing speed of the algorithm is improved, the redundant operation of the triangular facets still needs to be eliminated. Most algorithms are based on the slicing plane, which is used to search intersecting triangular facets. Huang et al. [10] took each triangular facet as the dominant factor. They used a hash table to store the relationship between triangular facets and intersecting planes. They searched the hash table to quickly search all slicing planes across the currently given triangular facet to achieve the effect of traversing each triangular facet only once. Zhang and Joshi [11] also proposed an efficient contour construction (ECC) algorithm by searching the intersecting slicing planes of triangular facets. Using the clockwise feature of the vertices of the triangle, the forward edge of the triangle is marked. Search the slicing plane intersected by each forward edge so that the intersecting edge and each slicing plane are calculated only once to reduce the calculation cost. King et al. [12] implemented a new algorithm based on ECC using edge matching, making the algorithm applicable to the topology of all models. Edge matching is an advanced sorting method that reduces the number of times each edge is sorted from n intersecting totals to two.

The triangular facets stored in the STL model are disordered, representing that it is not able to provide the algorithm with topological relationships between geometries when used as input. Therefore, Pan et al. [13] proposed an adaptive slicing algorithm to construct associations between triangles and their vertices in the triangle mesh. Zhang et al. [14] use a hash table to build the overall topology of the STL model in advance and reduce the slice search range by establishing a slice relation matrix. Tracing forward based on topological relation also reconstructs the information lost in the STL file when saving triangle mesh [15], [16]. Adaptive slicing is an algorithm that considers both machining efficiency and machining accuracy. It reduces the slice thickness in the area with more details and increases the layer thickness with fewer geometric features. There are many other adaptive slicing algorithms, such as [17], [18], and [19].

However, increasing or decreasing the thickness of the workpiece's slicing layer will hurt the deposition process above the current layer [20]. Therefore, adaptive slicing algorithms are generally suitable for fused deposition modeling (FDM) and are not commonly used in additive manufacturing techniques such as laser sintering (SLS), stereo photolithography (SL), or selective laser melting (SLM). Minetto et al. [21] showed that assembling discrete intersecting segments to construct the intersecting contour was time-consuming. Therefore, to improve the execution efficiency of the algorithm, in addition to pre-grouping sorting, the contour-construction step can also be improved. First, they judge and group the triangular facets according to the height of the slicing plane. Then, they store triangle-plane intersections through the hash table to realize the rapid reading of the intersections and efficient construction of the intersecting plane contour. But in fact, the hash function used in the hash table is expensive to compute, so Bhandari [22] chooses to use the graph structure instead of the hash table to store the desired intersections and uses depth-first search to traverse and read the intersections.

Most of the input of these slicing algorithms only consider the fixed Z-axis as the slicing direction, which has specific requirements for the placement position of the workpiece model, which inconveniences the process testing. In the past algorithms, the two steps of intersection calculation and contour construction were optimized without breaking the fixed two-step pattern. To calculate the triangle-plane intersections, each triangle will have two edges cross the slicing planes, one of which will be repeated as a shared edge, increasing the calculation amount of the algorithm. Because these algorithms get disordered discrete line segments after intersection calculation, line segment splicing is also needed to realize contour construction. However, there may be floating point errors when line segments are concatenated, resulting in the two line segments not matching and thus creating the wrong contour. In addition, according to the results of previous algorithms, it can be seen that the contour construction stage is time-consuming. Therefore, the algorithm proposed in this paper aims to solve the problems of limited slicing direction, high computation cost, and long computation time. Through the custom vector, the slicing direction can be more flexible, the redundant calculation can be reduced according to the topological relationship between the adjacent triangular facets and the shared edge, and the ordered intersections can be output directly to improve the execution efficiency of the algorithm.

III. METHODS

A. THE MAIN ALGORITHM

The main flow of the algorithm in this paper is shown below in Figure 1, which after reading the STL file groups all the triangular facets using the property that each layer of the cutting plane has the same height. Each layer of the cutting plane has its own set of triangular facets to eliminate redundant computations with other layers of triangular facets. The grouping is followed by a loop to iterate through the set of triangular facets for each layer. When processing the current layer triangular facet collection, a relational dictionary is constructed based on the rule that neighboring triangles have shared edges to compensate for the missing triangle mesh topology information from the STL file. As a result, the current layer cutting plane is used to intersect each shared edge only once to minimize the double counting of edges when following the triangle faces as the intersection unit. The loop is jumped out of at the end of the full layer traversal, and the choice is made to keep or reverse the orientation of the ordered intersections.

Algorithm 1 preprocessed the triangle mesh by binding the index number of each vertice through a dictionary so that the triangular facets list only stores the vertice index but not the specific coordinates. The algorithm's input is the vector *dir* for custom-given slicing direction, the layer thickness *gap*, an unordered list *Triangles*, and the list *Nodes* of the triangular facets, and the output is the list *Contour*. Discrete triangular facets are stored as items in list *Triangles*; each item contains three vertice indexes. The list *Nodes* stores all vertice coordinates of the triangle mesh, and we can find the triangle facest corresponding vertice coordinates by the serial number of *Nodes*. The list *Contour* stores each layer contour-group, and each layer contour group contains one or more polygons comprising triangle-plane intersections. Step 3 processes the input data according to the given slicing direction. Along the direction of vector *dir*, the vertices are sorted according to *Min*, *Med*, and *Max*, and the highest point and lowest point of the workpiece triangle mesh are obtained respectively as *mMax* and *mMin*, as detailed in section III-B. The STL file does not preserve the adjacency of the triangular facets in the triangle mesh; in fact, there is a shared edge between the adjacent triangular facets. Therefore, in step 4, the adjacent triangular facets are associated with their shared edges and stored in the dictionary *sTDic*. Each dictionary entry uses each edge in the triangle mesh as a key, and the two triangular facets that share this edge are stored as the corresponding value. However, the slicing plane of different layers usually intersects only some triangular facets in the triangle mesh, so the list *xT* and the variable *layers* are printed in this step.

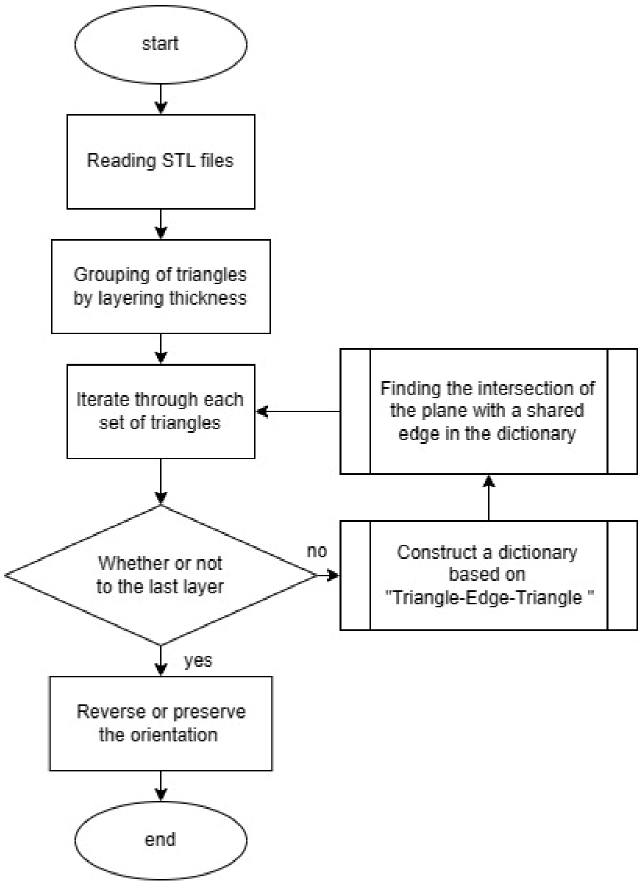

start
Reading STL files
Grouping of triangles by layering thickness
Iterate through each set of triangles
Finding the intersection of the plane with a shared edge in the dictionary
Whether or not to the last layer
no
Construct a dictionary based on "Triangle-Edge-Triangle "
yes
Reverse or preserve the orientation
end

FIGURE 1. Algorithm flow chart.

The list *xT* stores several sets of index of triangular facets that intersect only with the current slicing plane, while the variable *layers* represents the number of layers. This step is explained in detail in section III-C. In steps 5 through 16, the list *xT* is processed layer by layer, calculating their triangle-plane intersections. Step 7, according to the dictionary *sTDic*, finds the shared edges that intersect the current slicing plane layer by layer. These intersections are stored in the list *sGroup* in order, and these edges do not appear repeatedly, as detailed in section III-D. Therefore, the intersections obtained after the intersection calculation in step 10 are ordered and not repeated. Still, the orientation of the ordered intersections must be determined according to the variable *rev* returned in step 7. If the variable *rev* is true, the sequential numbers of the intersections must be reversed. Finally, the contour can be obtained directly in step 12; the inner contour is clockwise, and the outer contour is counterclockwise.

For step 9, when calculating the triangle-plane intersections, it is essentially to calculate the edge-plane intersection. And because a given custom slicing direction may be any vector, the slicing plane may no longer be perpendicular to the Z-axis. As shown in Figure 2, the edge-plane intersection is calculated when the slicing direction is not Z-axis. To facilitate the narration, the Angle of view of Figure 2.a is changed to observe the negative direction of the Y-axis

Algorithm 1

```
1: function optimal-slicing(dir, gap, Triangles, Nodes)
2:     p ← [ ][ ]; contour ← [ ];
3:     T,mMin, mMax ← Ref-Value-Init (dir, Triangles, Nodes);
4:     layers, sTDic, xT ← AdjTri-Corr-ShrdEdge(T, mMin, mMax);
5:     for i ∈[0,......, layers+1] do
6:         p ← RefLimit.Min + i∗gap;
7:         sGroup, rev ← Find-Intersct-Edge(sTDic, p, xT[i]);
8:         for each segs in sGroup do
9:             for each seg in segs do
10:                points ← Comput-Insection(seg, plane);
11:                if rev then reverse(points);
12:            end for
13:        end for
14:        p[i] ← points;
15:        contour ← p[i];
16:    end for
17:    return contour;
18: end function
```

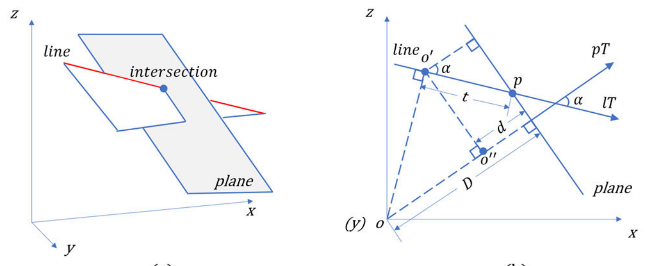


FIGURE 2. Calculate the intersection between the slicing plane of an arbitrary normal vector and an edge on a triangular facet.

to get Figure 2.b, and it can be seen that the cutting plane becomes a line. In Figure 2.b, the intersection p both on the plane and on the line, the distance from the plane to the origin o is known to be D, the normal vector pT of the plane and the direction vector lT of the line, need to find the point p. Essentially, the position of p is the distance t moved from the reference point o' along the direction of a line. First, project the point o onto the line as the reference point o', then the point o' onto the normal vector pT of the plane to obtain the projection length oo''. Then subtract oo'' by D to get d, and the dot product of the normalized vectors pT and lT gives $cos\alpha$. Finally, t is obtained according to the triangle similarity principle and the pythagorean theorem, and then it is substituted into the line to get the intersection p.

B. REFERENCE VALUE INITIALIZATION

Because the given slicing direction may not be along the Z-axis, if the original vertice coordinate value is compared and judged, it will only get the wrong results and affect the accuracy of the contour. Therefore, before the core work of the slicing algorithm, the Ref-Value-Init(algorithm 2) process needs to be executed to initialize some reference values, and the reference values enable the algorithm to correctly judge the comparison relationship, which appears in step 3 of algorithm 1. The input to this procedure is the slicing direction *dir*, the unprocessed list *Triangles*, and the

Algorithm 2

```
1: function Ref-Value-Init (dir, Triangles, Nodes)
2:     N ← [] ; T ← [];
3:     for each n ∈ Nodes do
4:         N ← n dot dir;
5:     end for
6:     for each t ∈ Triangles do
7:         T ← (t.item1, t.item2, t.item3);
8:     end for
9:     for i ∈ [0,...,T.length] do
10:        tri ← T[i];
11:        v←N[tri.index[0]];v←N[tri.index [1]]; v2←N[tri.index [2]];
12:        if v0 > v1 then
13:            max ← v0; min ← v1;
14:            T[i].Max ← tri.Index[0];
15:            T[i].Min ← tri.Index[1];
16:            T[i].Mid ← tri.Index[2];
17:        else
18:            max ← v1; min ← v0;
19:            T[i].Max ← tri.Index[1];
20:            T[i].Min ← tri.Index[0];
21:            T[i].Mid ← tri.Index[2];
22:        end if
23:        if max < v2 then
24:            max ← v2;
25:            T[i].Mid ← tri.Max; T[i].Max ← tri.Index[2];
26:        end if
27:        if min > v2 then
28:            min ← v2;
29:        T[i].Mid ← tri.Min; T[i].Min ← tri.Index[2];
30:        end if
31:        T[i].Refmin ← min; T[i].Refmax ← max;
32:        if mMin > min then mMin ← min;
33:        if mMax < max then mMax ← max;
34:    end for
35:    return T, mMin, mMax;
36: end function
```

list *Nodes*. The output of this process is the list T, where each vertice of all the triangles is marked, and the highest and lowest vertices of the triangle mesh are obtained.

From Step 3 to Step 5, the process traversed the list *Nodes* to read all vertices coordinates. They are dotted with the vector *dir* to give a list N containing the reference values of all vertices of the triangle mesh, that is, the vertices of the triangular facet. From Step 6 to Step 8, traverse the list *Triangles* to retrieve the triangular facet data to list T, where each item comprises the index of three vertices from which a specific coordinate value can be extracted from the list *Nodes*. From Step 9 to Step 33, sort the three vertices of each triangular facet, find the maximum, minimum, and middle values of the vertices along the vector *dir*, and label them *Max*, *Min*, and *Med*, as shown in the triangle e in Figure 3. For example, *T[i].max* stores the index of the highest vertice of the ith triangle. In addition, the smallest and largest vertices in all vertices are taken as the lowest point *mMin* and the highest point *mMax* of the whole triangle mesh, respectively.

C. ADJACENT TRIANGLES AND A SHARED EDGE

These facets in a triangle mesh are adjacent, and two adjoining triangular facets share one edge. In addition, each triangle

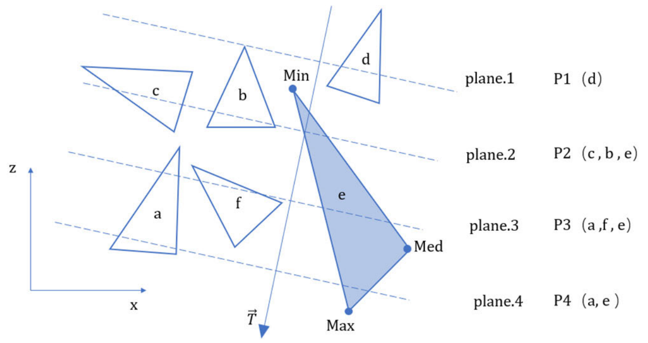


FIGURE 3. Triangular facets are grouped according to the slicing planes.

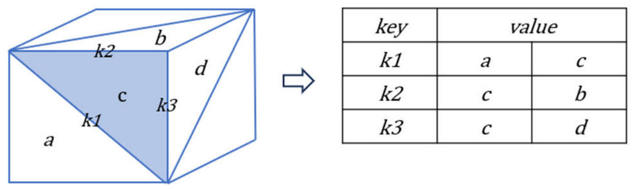
has only two edges intersecting the slicing plane; if the usual algorithm treats the triangular facet as an independent individual, many edges will be calculated twice. Therefore, treating these triangular facets as a related whole and the slicing process as an intersecting calculation between these shared edges and the slicing planes rather than between disordered triangular facets and the planes is necessary. The method AdjTri-Corr-ShrdEdge (Algorithm3) is used to find the relationship between adjacent triangles and their shared edges and store it so that the contour can be output directly in the intersection calculation phase. It appears in Step 4 of Algorithm 1. The input to this process is the list T, the variables *mMin* and *mMax*. The list T stores triangles with ordered vertices, *mMin* and *mMax* representing the lowest and highest points of the model, respectively. The output of this process is list *xT*, dictionary *sTDic*, and variable *layers*. List *xT* stores the triangular facets intersecting the current plane layer by layer, dictionary *sTDic* stores these shared edges and their corresponding triangular facet, and variable *layers* represents the number of slicing planes. There are many facets in the triangle mesh, but when the plane's height increases along the slicing direction, not all triangular facets intersect with the current plane. Therefore, from step 4 to step 10, the triangular facets are grouped into the list *xT* according to the height of the intersecting plane, and each triangular facet is labeled in ascending order according to traversing the list T. For example, *xT[i]* stores the index of all triangular facets intersecting the plane in which the index is i. A triangular facet may cross multiple planes, and in step 5, divide the difference between the triangle's lowest vertice and the model's lowest vertice by the layer thickness; the index of the lowest and highest layers in the intersecting slicing planes can be obtained. As shown in Figure 3, the Min-Max edge of the triangular facet e crosses three planes along the slice direction *dir*, so when traversing it, it is inserted into the groups *P1*, *P2*, and *P3*.

Then, from step 12 to step 25, traverse the list *xT*, creating a relational dictionary of the shared edge and adjacent triangular facets. The shared edge is stored as a key, and the two adjacent triangles are stored in value. First, from step 13 to step 16, find the current triangular facet from list T according to the index number *tIndex*, and then label its three edges

Algorithm 3

```
1: function AdjTri-Corr-ShrdEdge(T, mMin, mMax)
2:    layers ← ( mMax – mMin )/gap;
3:    xT[layers+1][ ] ← [ ][ ]; xTLth[layers+1] ← [0]; tIndex ← 0;
4:    for each t ∈ T do
5:            btm←(t.Refmin- mMin)/gap; top←(t.Refmax- mMin)/gap;
6:        for i ∈ [btm,......,top] do
7:            xT[i][xTLth[i]] ← tIndex; xTLth[i] ++;
8:        end for
9:      tIndex ++;
10:       end for
11:       value ← [ ]; sTDic ← (, value); triIndex ← 0;
12:       for i ∈ [0,......, layers+1] do
13:         tIndex ← xT[i][ xTLth [i]]; tri ← T[tIndex];
14:         k1 ← getKey(tri.Index[0], tri.Index[1]);
15:         k2 ← getKey(tri.Index[1], tri.Index[2]);
16:         k3 ← getKey(tri.Index[2], tri.Index[0]);
17:         // insert triangle into the dictionary entry with key k1
18:         if sTDic -contain-Key (k1) then sTDic[k1] [1] ← tIndex;
19:         else
20:             value[0] ← tIndex; sTDic.Add(sKey1, value);
21:         end if
22:         // end insert
23:         repeatinsert operation for k2 and k3
24:      triIndex ++;
25:      end for
26:      return layers, xT, sTDic;
27: end function
```

as *k1*, *k2*, and *k3*. Then, the dictionary *sTDic* is operated. If edge *k1* is not in the dictionary, a new entry is created, *k1* is inserted into it as a key, and the index of the current triangular facet is stored as *value[0]*. If *k1* is in the dictionary, add triangular facets directly into the entry with key *k1* as *value [1]*. Continue to process the *k2* and *k3* edges according to this principle until the processing of the current triangular facet is completed, as shown in Figure 4. Similarly, loop to the next triangular facet and do the same until all triangles and their edges in the list *xT* are stored in the dictionary *sTDic* as "two adjacent triangles and one edge" correspondence.



key	*value*	
k1	*a*	*c*
k2	*c*	*b*
k3	*c*	*d*

FIGURE 4. Creates a dictionary structure for triangle-edge.

D. FIND INTERSECTING SHARED EDGES

The process of Find-Intersec-Edge (Algorithm 4) is to find all shared edges intersecting slicing planes in the whole triangle mesh in order, which appears in step 7 of Algorithm 1. The input to this procedure is the dictionary *sTDic* that stores the correspondence between adjacent triangles and shared edges, the current slicing plane parameter p, and the list *xT[i]* represents all intersecting triangular facets of the ith layer. The output of this process is the list *sGroup*,which means

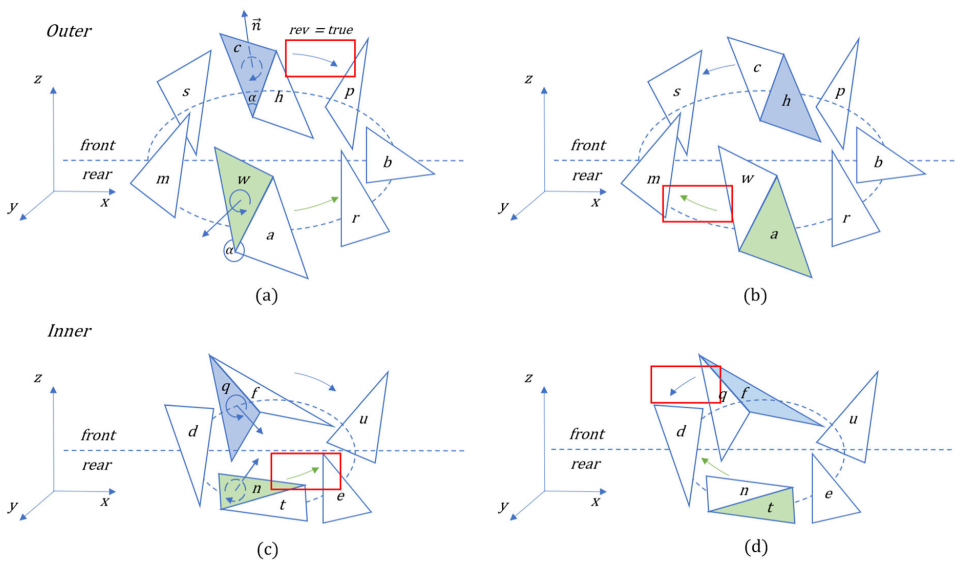


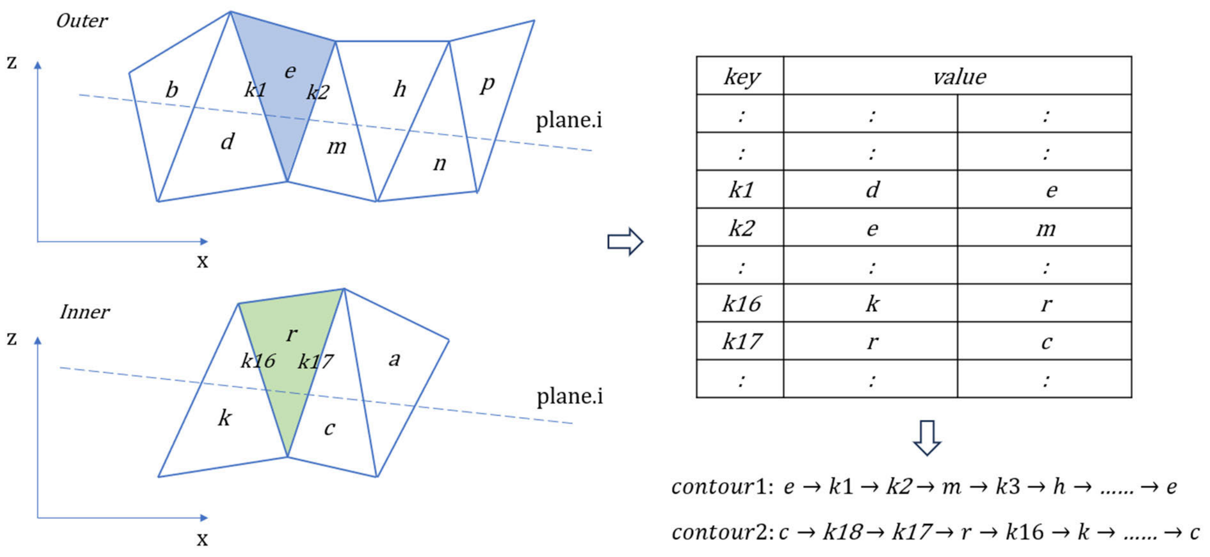
FIGURE 5. To adjust the orientation of the ordered intersection.

Algorithm 4

```
1: Function find-intersecting-edges(sTDic, p, xT[i])
2:      sGroup ← [ ]; tri ← xT[i]; uDic ← (, );
3:      //Mark the triangles
4:      mk ← -1;
5:      For each i ∈ tri do
6:          if i = -1 then continue;
7:          if uDic-notContain-key (i) then mk ← i; break;
8:          else i ← -1;
9:          end if
10:     end for
11:     //end Mark
12:     while mk != -1 do
13:         t ← T[mk]; e0 ← (N[t.Min], N[t.Max]); s.Add(e0);
14:         v0 ← t.Mid; v1 ← t.Mid.RefValue <= p ? t.Max : t.Min;
15:         e ← (N[t.Mid], N[t.Min]);
16:         angle ← e0 rotateTo e; rev ← (angle < $\pi$);
17:         //find adjacent triangle based on segment
18:         e1 ← (v0, v1); s.Add(e1); uDic.Add(mk, false);
19:         key ← getKey(e1);
20:         mkArr ← sTDic[key]; mk ← mkArr[0];
21:         if uDic-Contain-Key(mk) then mk ← mkArr [1];
22:         //end find
23:             while (uDic-notContain-Key(mk)) do
24:                 t ← T[mk]; sKey ← getKey(t.Min, t.Max);
25:                 if sKey = key then
26:                     v0 ← t.Mid;
27:                     v1 ← t.Mid.RefValue <= p ? t.Max : t.Min;
28:                 else
29:                     v0 ← t.Min; v1 ← t.Max;
30:                 end if
31:                 repeatfind adjacent triangle
32:             end while
33:             if s.cnt >2 then sGroup.Add(s);
34:             repeat mark the triangle
35:     end while
36:     return sGroup, rev;
37: end function
```

edges intersecting the plane and will be the input of the next step to calculate edge-plane intersections. From step 3 to step 11, the triangular facet is marked whether it has traversed through the dictionary *uDic*. If *uDic* does not contain the currently judged triangular facet, the value of *mk* is the index of the triangular facet; otherwise, the value is -1. If the current layer has untraversed triangular facets, you can use a random triangular facet as a seed facet for steps 12 to 32. Because in the two edges of the triangular facet crossing the slicing plane, the highest vertice *Max* and the lowest vertice *Min* are the endpoints of the one side. So, step 13 adds the Min-Max edge of the seed facet to the current contour list **s** as the first intersecting edge *e0* found. Then, look for the second intersecting edge of the seed facet and determine whether the slicing plane is above or below the middle vertice *Med* in step 14 along the slicing direction. If the plane is above *Med*, the endpoints of the second intersecting edge *e1* are vertices *Max* and *Med* of the triangular facet; otherwise, they are vertices *Min* and *Med*. After being stored in list **s**, the seed facet is processed, and the value of *mk* about the triangular facet in list *uDic* is updated to false.

Before traversing the next triangular facet, it is necessary to judge by step 15 whether the edge Min-Max of the seed triangular facet is rotated counterclockwise around the normal vector to the edge Min-med at an angle greater than 180°. If the angle is less than 180°, *rev* is true; otherwise, *rev* is false. As shown in Figure 5, the position of the selected seed facet affects the value of the Angle α. The positions of these triangular facets crossing the plane can be divided into front and rear along the negative Z-axis. Because the first intersecting edge of the seed facet is always set to the Min-Max edge, the position of the triangular facet above or below determines whether the search orientation of the intersecting edge is clockwise or counterclockwise. In addition, when the seed facet belongs to two different dependent cases of the inner or outer contour, its normal vector is used as the rotation axis of the edge Min-Max to the edge Min-Med, which will



key	*value*	
:	:	:
:	:	:
k1	*d*	*e*
k2	*e*	*m*
:	:	:
k16	*k*	*r*
k17	*r*	*c*
:	:	:

*contour*1: $e \rightarrow k1 \rightarrow k2 \rightarrow m \rightarrow k3 \rightarrow h \rightarrow \ldots\ldots \rightarrow e$

*contour*2: $c \rightarrow k18 \rightarrow k17 \rightarrow r \rightarrow k16 \rightarrow k \rightarrow \ldots\ldots \rightarrow c$

FIGURE 6. Find the shared edges associated with the slice contour and compute the intersections.

also lead to different results. Both Figure 5.a and Figure 5.b show the outer contour, while Figure 5.c and Figure 5.d show the inner contour. For example, in Figure 5.a, the triangular facet *c* in the front of the outer contour is selected as the seed facet, and its normal vector *n* faces the negative direction of the Y-axis. The edge Min-Max rotates counterclockwise around the rotation axis *n* to the edge Min-Med, and the Angle α is less than 180°, so the bool value of *rev* is updated to true. However, when the triangular facet w in the rear outer contour is selected as the seed facet, the Angle α is greater than 180^°, and the bool value of *rev* remains false. Similarly, in the case of the other three diagrams, the *rev* will also be updated to true, marked with a red box in the diagram. The variable *rev* in this step is used to correct the order of intersections in step 11 of Algorithm 1, ensuring that the orientation of the contour formed by the ordered intersections is stored according to the rule of the outer contour is counterclockwise, and the inner contour is clockwise. From step 16 to step 21, find the adjacent triangular facet of the seed facet as the next triangular facet needs to be traversed, and look for the value array in the dictionary *sTDic* according to key *e1*. If the triangular facet in *value[0]* does not exist in *uDic*, it means that the triangular facet has not been traversed, then fetch it, otherwise, choose the triangular facet in *value [1]*.

From step 22 to step 29, find intersecting edges according to the principle of edge-triangle-edge until you return to the seed facet. From step 23 to step 28, the steps are to find the next intersecting edge. At this point, the shared edge *e1* has been processed as the first intersecting edge of the second triangular facet, and only its second intersecting edge needs to be found. You can judge it based on the previous intersecting edge. According to Step 14, to determine which of the two intersecting edges acts as the second edge to traverse, the Min-Max edge appears as the first or second intersecting edge. Starting with the second triangular facet in the current traversal, only one intersecting edge per triangle needs to be processed to mark the value of *mk* as false in *uDic*, that is, to repeat steps from step 16 to step 21 in step 29. Completing the above loop represents the last traversal back to the seed facet, which means that a closed contour can be obtained based on these intersecting edges, so it is stored in the list *sGroup* in step 31. Because there may be multiple contours in a slicing plane, repeat steps from step 3 to step 11 at step 32 until all intersecting triangular facets are traversed. The above process is shown in Figure 6. There are two contours, *contour1* and *contour2*, on the slicing plane of layer *i*, and the intersection edges related to the two contours are found in turn according to the relation dictionary. For example, in contour1, suppose that the triangular facet **e** is the seed facet, its first intersecting edge is *k1*, and its second intersecting edge is *k2*. Then, according to *k2*, the triangular facets sharing it are *e* and *m*. Since **e** has already traversed, we take out *m*. Then, find the second intersecting edge *k3* of the triangular facet *m*, and so on to find *h*, ending the search when the loop returns to the triangular facet *e*, and all the intersecting edges related to *contour1* have been found.

IV. EXPERIMENT

To evaluate the execution efficiency of this algorithm, 13 STL models are used as inputs for experiments, and the running time required for slicing different models is output. These 13 models are the criteria mentioned in Minetto's classical algorithm, as shown in Table 1. With the increase of sequence number, the complexity and size of the model increase gradually.

The algorithm is implemented in C# language, the running time is tested on Intel core i7 devices, and the running environment system is Windows 11. The test parameters include layer thickness and slicing direction. The algorithm slices some models(Liver, Demon, Sphenoid, Tesla, Sphere, Soldier) in Table 1 and visualizes the output results.

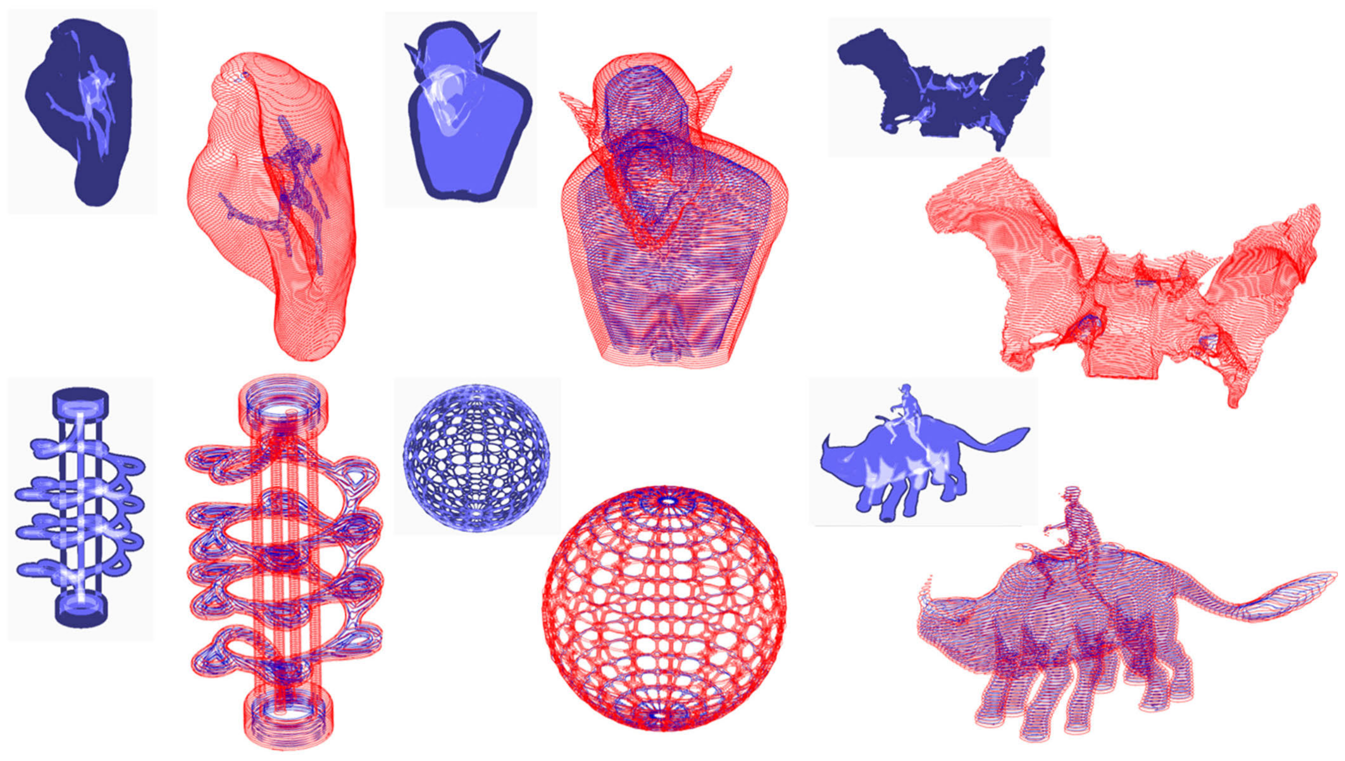


FIGURE 7. Visual slicing results.

TABLE 1. Properties of the STL model.

STL name	Number of triangles	Size (MB)
01.Liver	38,142	6.4
02.Femur	42,150	11.4
03.Bunny	270,021	12.9
04.Demon	935,236	44.6
05.Sphenoid	983,134	47.0
06.Rider	1281,950	61.1
07.Tesla	2296,928	110.0
08.Sphere	3060,992	146.0
09.Soldier	3394,035	161.8
10.Bear	6248,232	298.0
11.Warrior	8852,207	422.1
12.Robot	12,274,288	585.3
13.Skull	16,668,257	795.0

To facilitate the observation of the division of internal and external contours, the slicing direction is uniformly selected as the Z-axis and the layer thickness is set to 1.2 mm. The outer contour is shown in red, the inner contour is shown in purple, each X-ray view of the STL models is shown on the upper left, and the slice results are shown on the lower right, as shown in Figure 7.

To compare with other algorithms, the layer thickness is set to 0.032mm, and the slice direction is selected as the positive direction of the Z-axis. The Minetto algorithm was chosen for comparison algorithms because it is a representative slicing algorithm. The algorithm in the CGAL library was selected because of its high visibility and recognition. In the configuration of the algorithm in this paper, the other two algorithms are also used to slice 13 STL models. For comparison, the algorithm in this paper is also fixed on the Z-axis as the slicing direction, like the other algorithms. The comparison of the running time required by the three algorithms is shown in Table 2. It can be seen from the result data that although the geometric features of the 13 STL models are different, the proposed algorithm has improved the execution efficiency compared with other algorithms, indicating that the optimization calculation effect is remarkable. The required runtime was reduced by 91% on the model liver for Minetto and 67% on the model rider for CGAL.

Previous slicing algorithms would fix the Z-axis for calculation and optimization, but the algorithm in this paper can customize any vector as the slicing direction. Taking the STL model rabbit as an example, the visualization of slicing results is shown in Figure 8. The same rabbit model is sliced along three different vector directions, with the outline of the slice attached to the model at the bottom left and the representation of the vector direction at the top right. Regarding the setting of the given parameters, the layer thickness is set to 1.0mm and the running time is 122ms, 118ms, and 215ms from

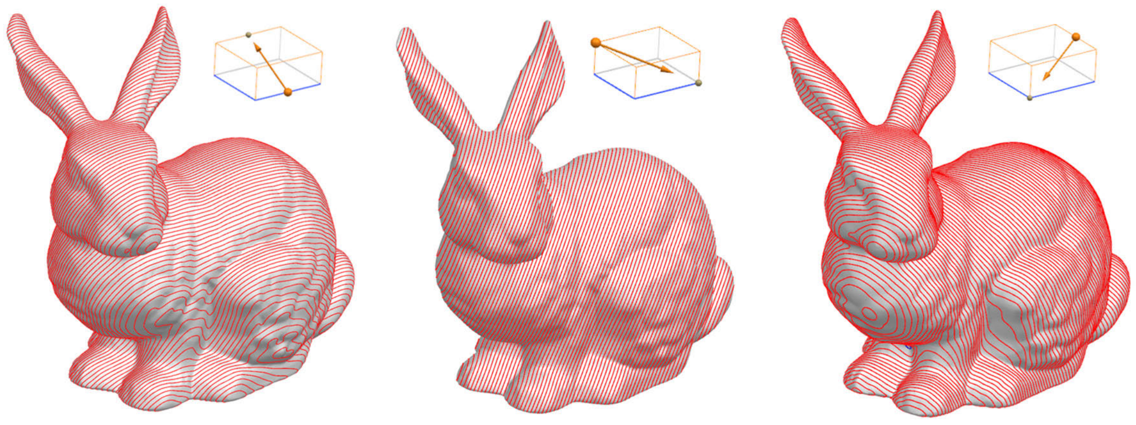


FIGURE 8. Slice the model in a customized arbitrary direction.

TABLE 2. Compare with other algorithms.

Model	Minetto(ms)	CGAL(ms)	Proposed(ms)
01.liver	2304	209	**194**
02.femur	1156	118	**101**
03.bunny	1712	1022	**292**
04.demon	6849	4660	**1402**
05.sphenoid	4399	3103	**915**
06.rider	2364	4136	**775**
07.tesla	11528	7262	**3040**
08.sphere	19850	10606	**4642**
09.soldier	13484	9353	**4745**
10.bear	72884	24381	**13391**
11.warrior	4035	4163	**3824**
12.robot	9447	9026	**7701**
13.skull	323709	289715	**109317**

left to right. The results show that different slice directions will change the number of triangle faces to be processed in each layer, and will also affect the running time of the algorithm, but will not affect the stability and effectiveness of the proposed algorithm.

V. CONCLUSION

Compared to previous algorithms, the algorithm in this paper improves the execution efficiency by optimizing the computation. Whether it is a simple small-size model or a complex model with more triangular facets, the slices use less runtime, representing more efficient execution. Evaluation from the STL model dataset reveals up to 91% time reduction. In addition, when using arbitrary vectors as the slicing direction, the accuracy of layering the model is still maintained and the visualization results show that the contours are tightly connected to the model. The design of the algorithmic logic breaks the two-step approach while the output results remain valid and stable, providing more possibilities for the future development of slicing algorithms. In the future, we will incorporate parallel computing mechanisms into our algorithms, transforming sequential processing of each layer into simultaneous processing to achieve faster efficiency.

APPENDIX A
ALGORITHMIC VARIABLES SUMMARY

dir	The given slicing direction
gap	The layer thickness
Triangles	The list of stored unordered triangular facets
Nodes	The vertices of all triangular facets
n	Normal vector of a triangular facet
contour	The list stores unordered intersection points
Min	The vertex of the triangle with the lowest position on the given vector
Med	The vertex of the triangle positioned in the middle on a given vector
Max	The vertex of the triangle with the highest position on the given vector
mMax	Highest position of the triangle mesh of the fitted artifacts
mMin	Lowest position of the triangle mesh of the fitted artifacts
sTDic	The dictionary that stores adjacent triangle pairs and their shared edges
layers	Number of layers to be sliced
xT	List of triangular facets intersecting the current cutting plane
sGroup	the list stores ordered intersection points
rev	Flags that determine whether the order of intersections should be reversed or not
uDic	Dictionary to record traversed triangular facets
mk	Flags for whether the triangular facets has been traversed

REFERENCES

[1] P. Kulkarni, A. Marsan, and D. Dutta, "A review of process planning techniques in layered manufacturing," *Rapid Prototyping J.*, vol. 6, no. 1, pp. 18–35, Mar. 2000, doi: 10.1108/13552540010309859.

[2] I. Gibson, D. Rosen, and B. Stucker, "Introduction and basic principles," in *Additive Manufacturing Technologies*, 3rd ed. New York, NY, USA: Springer, 2010, pp. 20–35. [Online]. Available: https://link.springer.com/book/10.1007/978-3-030-56127-7

[3] Y. Qin, Q. Qi, P. J. Scott, and X. Jiang, "Status, comparison, and future of the representations of additive manufacturing data," *Computer-Aided Design*, vol. 111, pp. 44–64, Jun. 2019, doi: 10.1016/j.cad.2019.02.004.

[4] P. Mohan Pandey, N. Venkata Reddy, and S. G. Dhande, "Slicing procedures in layered manufacturing: A review," *Rapid Prototyping J.*, vol. 9, no. 5, pp. 274–288, Dec. 2003, doi: 10.1108/13552540310502185.

[5] W. Zha and S. Anand, "Geometric approaches to input file modification for part quality improvement in additive manufacturing," *J. Manuf. Processes*, vol. 20, pp. 465–477, Oct. 2015, doi: 10.1016/j.jmapro.2015.06.021.

[6] P. Nayyeri, K. Zareinia, and H. Bougherara, "Planar and nonplanar slicing algorithms for fused deposition modeling technology: A critical review," *Int. J. Adv. Manuf. Technol.*, vol. 119, nos. 5–6, pp. 2785–2810, Jan. 2022, doi: 10.1007/s00170-021-08347-x.

[7] C. F. Kirschman and C. C. Jara-Almonte, "A parallel slicing algorithm for solid freeform fabrication process," Tech. Rep., 1992, doi: 10.15781/T2NS0MF5K.

[8] K. Tata, G. Fadel, A. Bagchi, and N. Aziz, "Efficient slicing for layered manufacturing," *Rapid Prototyping J.*, vol. 4, no. 4, pp. 151–167, Dec. 1998, doi: 10.1108/13552549810239003.

[9] R. Tian, S. Liu, and Y. Zhang, "Research on fast grouping slice algorithm for STL model in rapid prototyping," *J. Phys., Conf. Ser.*, vol. 1074, Sep. 2018, Art. no. 012165, doi: 10.1088/1742-6596/1074/1/012165.

[10] X. Huang, Y. Yao, and Q. Hu, "Research on the rapid slicing algorithm for NC milling based on STL model," in *Proc. Asia Simul. Conf.*, 2012, pp. 263–271. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-34387-2_30

[11] Z. Zhang and S. Joshi, "An improved slicing algorithm with efficient contour construction using STL files," *Int. J. Adv. Manuf. Technol.*, vol. 80, nos. 5–8, pp. 1347–1362, Sep. 2015, doi: 10.1007/s00170-015-7071-9.

[12] B. King, A. Rennie, and G. Bennett, "An efficient triangle mesh slicing algorithm for all topologies in additive manufacturing," *Int. J. Adv. Manuf. Technol.*, vol. 112, nos. 3–4, pp. 1023–1033, Dec. 2020, doi: 10.1007/s00170-020-06396-2.

[13] X. D. Pan, K. Chen, Z. Y. Zhang, D. F. Chen, and T. T. Li, "Adaptive slicing algorithm based on STL model," *Appl. Mech. Mater.*, vol. 288, pp. 241–245, Feb. 2013, doi: 10.4028/www.scientific.net/amm.288.241.

[14] Z. Zhang, Z. Zhu, Y. Feng, and R. Li, "Improvement and optimization of slicing algorithm for rapid prototyping manufacturing," *Adv. Mech. Eng.*, vol. 15, no. 3, Mar. 2023, Art. no. 168781322311532, doi: 10.1177/16878132231153284.

[15] S. Rock and M. Wozny, "Utilizing topological information to increase scan vector generation efficiency," in *Proc. Solid Freeform Fabr. Symp.*, Austin, TX, USA, 1991, pp. 28–36.

[16] S. Mcmains and C. Séquin, "A coherent sweep plane slicer for layered manufacturing," in *Proc. 5th ACM Symp. Solid Model. Appl.*, New York, NY, USA, Jun. 1999, pp. 285–295.

[17] W. Wang, H. Chao, J. Tong, Z. Yang, X. Tong, H. Li, X. Liu, and L. Liu, "Saliency-preserving slicing optimization for effective 3D printing," *Comput. Graph. Forum*, vol. 34, no. 6, pp. 148–160, Sep. 2015, doi: 10.1111/cgf.12527.

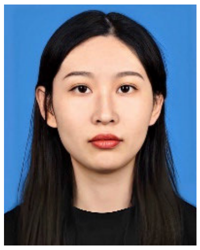
[18] B. Hu, G. Jin, and L. Sun, "A novel adaptive slicing method for additive manufacturing," in *Proc. IEEE 22nd Int. Conf. CSCWD*, 2018, pp. 218–223. [Online]. Available: https://ieeexplore.ieee.org/document/8465247

[19] F. Wasserfall, N. Hendrich, and J. Zhang, "Adaptive slicing for the FDM process revisited," in *Proc. 13th IEEE Conf. CASE*, 2017, pp. 49–54. [Online]. Available: https://ieeexplore.ieee.org/document/8256074

[20] Q. Li and X. Y. Xu, "Self-adaptive slicing algorithm for 3D printing of FGM components," *Mater. Res. Innov.*, vol. 19, no. sup5, pp. S5-635–S5-641, May 2015, doi: 10.1179/1432891714z.0000000001167.

[21] R. Minetto, N. Volpato, J. Stolfi, R. M. M. H. Gregori, and M. V. G. da Silva, "An optimal algorithm for 3D triangle mesh slicing," *Comput.-Aided Design*, vol. 92, pp. 1–10, Nov. 2017, doi: 10.1016/j.cad.2017.07.001.

[22] S. Bhandari, "A graph-based algorithm for slicing unstructured mesh files," *Additive Manuf. Lett.*, vol. 3, Dec. 2022, Art. no. 100056, doi: 10.1016/j.addlet.2022.100056.



HAN LIU received the B.E. degree in computer science and technology from Daqing Normal University, Daqing, China, in 2014, and the M.E. degree in computer science and technology from Shenyang University of Technology, Shenyang, China, in 2019, where she is currently pursuing the Ph.D. degree in mechanical engineering. Her current research interests include optimization algorithms, computational geometry, and their applications in the field of additive manufacturing.

FEI XING received the B.E. degree in mechanical and electronic engineering from Northeastern University, Shenyang, China, in 2003, and the Ph.D. degree in mechanical and electronic engineering from Shenyang Institute of Automation Chinese Academy of Sciences, Shenyang, in 2009. He is currently a Professor with Shenyang University of Technology. His current research interests include addictive manufacturing, optimization algorithms, and laser cladding forming.