A Geometric Approach for Computing the Kernel of a Polyhedron

T. Sorgente[1] and S. Biasotti[1] and M. Spagnuolo[1]

[1]Istituto di Matematica Applicata e Tecnologie Informatiche "E. Magenes" - CNR Italy
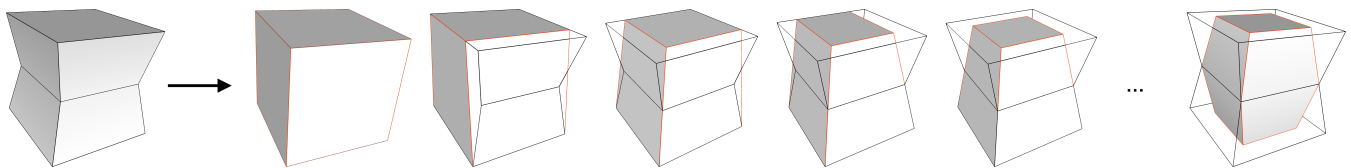

Figure 1: *Pipeline of the kernel computation for a polyhedron. At first step, we compute the Axis Aligned Bounding Box (AABB) of the polyhedron; then, we iterate on each face f of the polyhedron (black edges) and cut AABB with the plane induced by f (red edges).*

Abstract
We present a geometric algorithm to compute the geometric kernel of a generic polyhedron. The geometric kernel (or simply kernel) is defined as the set of points from which the whole polyhedron is visible. Whilst the computation of the kernel for a polygon has already been largely addressed in the literature, less has been done for polyhedra. Currently, the principal implementation of the kernel estimation is based on the solution of a linear programming problem. We compare against it on several examples, showing that our method is more efficient in analysing the elements of a generic tessellation. Details on the technical implementation and discussions on pros and cons of the method are also provided.

CCS Concepts
• *Computing methodologies* → *Volumetric models;*

1. Introduction

The concept of *geometric kernel* of a polygon, a polyhedron, or more generally of a shape, is a pillar of computational geometry. Intuitively, the kernel of a close, geometric shape S is the locus of the points internal to S from which the whole shape S is visible. This concept is particularly interesting when applied to non convex polytopes, as for convex shapes the kernel coincides with the shape itself.

In the simplest scenario, that is if the shape is a polygon, the standard way of computing its kernel is by intersecting appropriate half-planes generated from its edges. This problem has been tackled since 70s, when [SH76] presented an efficient algorithm that performed the kernel computation in $O(e \log e)$ operations, where e is the number of edges of a polygon, as the intersection of e half-edges. After that, an optimal algorithm able to run in $O(n)$ operations, being n the number of vertices of the polygon, has been proposed in [LP79]. Up to our knowledge, computational tools and libraries like *Boost* [Boo21], *Geogram* [LF15], *CGAL* [FP09], or *Libigl* [JP17] implement routines to compute intersections between polygons and planes, which can be used to estimate the kernel. In

the first attempts, for example in [PS85], the extension of the problem to the 3D case was treated only from a theoretical point of view. Starting from this perspective, the natural approach of extending of the 2D method (which we call the *geometric approach)* was soon dismissed as unattractive for computational reasons. It was replaced by a new approach (which we call *algebraic)* which makes use of linear algebra and homogeneous coordinates, and that is the state of the art for computing 3D kernels currently implemented by libraries like CGAL.

During years, the polygon kernel computation has become popular to address several problems based on simple polygon analysis, such as star-component decomposition and visibility algorithms that are of interest in robotics, surveillance, geometric modeling, computer vision and, recently, in the emerging field of additive manufacturing [DAB18]. Today, the geometric kernel of a polytope is a pivotal information for understanding the geometrical quality of an element in the context of finite elements analysis. While in the past years finite elements methods were only designed to work on convex elements like triangles/tetrahedra or quadrangles/hexahedra [Cia02], recent and more complex methods like the Mimetic Fi-

nite Difference Method [LMS14], the Virtual Elements Method [BBC*13], the Discontinuous Galerkin Mehod [CKS12] or the Hybrid High Order Method [DD19] are able to deal with non convex polytopes. This enrichment of the class of admissible elements led researchers to further investigate the idea of the geometric quality of a polytope, and to define quality measures and metrics for the elements of a mesh [ABB*21; SBMS21]. In this setting, the geometric kernel is often associated with the concepts of *shape regularity* and *star-shapedness* of an element. For example, as analyzed in [SPC*21], most of the error estimates regarding the VEM (but the same holds for other polytopal methods) are based on the theory of polynomial approximation in Sobolev spaces, assuming the starshapedness of the elements [SB08; DS80]. As a consequence there are a number of sufficient geometrical assumptions on the computational domain for the convergence of the method, which require an estimate of the kernel. Mesh quality measures/metrics/indicators require to compute the kernel of thousands of polytopes, each of them with a limited number of faces and vertices, in the shortest possible time [SBMS21].

In this paper we define an algorithm for the implementation of the geometric approach to the computation of the kernel of a polyhedron, and empirically show how this approach can significantly outperform the algebraic one when applied in the context of finite elements methods, with elements having a limited number of faces and vertices.

The paper is organized as follows. In Section 2 we introduce and define the concept of kernel of a polytope in dimension 2 and 3. In Section 3 we detail the algorithm for the construction of the kernel of a polyhedra. In Section 4 we exhibit some examples of computed kernels and analyze the performance of the algorithm, also with comparisons with an implementation of the algebraic approach. In Section 5 we sum up pros and cons of the algorithm and draw some conclusions.

2. Terminology and preliminary concepts

Let us introduce some basic concepts, useful to the computation of the kernel of a polyhedron. Following the notation adopted in [PS85], a *polyhedron* is defined by a finite set of plane polygons such that every edge of a polygon is shared by exactly one other polygon and no subset of polygons has the same property. The vertices and the edges of the polygons are the *vertices* and the *edges* of the polyhedron; the polygons are the *faces* of the polyhedron. In this work we only consider *simple* polyhedra, which means that there is no pair of nonadjacent faces sharing a point.

A polyhedron P is *convex* if, for any two points p_1 and p_2 in P, the segment (p_1, p_2) is entirely contained in P. It can be shown that the intersection of convex polyhedra is a convex polyhedron. Two points p_1 and p_2 inside P are said to be *visible* if the line segment connecting p_1 and p_2 does not intersect with the exterior of the P. It is easily seen that any two points inside a convex polyhedron are visible. The *kernel* of a P is the set of points from which all points inside P are visible. Some polyhedra may not have a kernel, or we also say their kernel is empty. The polyhedron P is called *starshaped* if there exists a sphere completely contained in its kernel. A polyhedron is not star-shaped if its kernel is empty.

In Fig. 2 we present a parametric polyhedron shaped like a tent, with the parameter regulating the height of the "entrance". As the parameter increases, the set of points from which the whole polyhedron is visible becomes smaller, and so does the kernel. The last example of Fig. 2 is not star-shaped anymore, i.e. the kernel is empty.
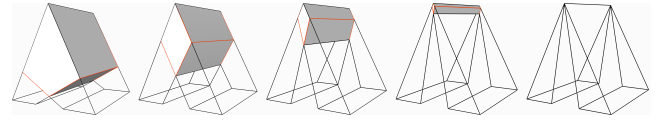

Figure 2: *Evolution of the kernel (in red) in a sequence of parametric polyhedra.*

In Section 3 we will make use of the concept of *signed distance*. Given a plane p with unit normal vector $\boldsymbol{n}$ and a point $\boldsymbol{p}$ on it, the signed distance between a point $\boldsymbol{x}$ and the plane is given by the scalar product $d(\boldsymbol{x}, p) = \boldsymbol{n} \cdot (\boldsymbol{p} - \boldsymbol{x})$. We say that $\boldsymbol{x}$ is strictly (weakly) below p if $d(\boldsymbol{x}, p) < 0 \ (\leq 0)$, $\boldsymbol{x}$ is strictly (weakly) above if $d(\boldsymbol{x}, p) > 0 \ (\geq 0)$, and $\boldsymbol{x}$ is on p if $d(\boldsymbol{x}, p) = 0$.

2.1. Geometric vs algebraic approach

The state of the art algorithm in the 2D case follows a geometric approach: the kernel of a polygon is found as the intersection of half-planes originating from its edges. We use the term "geometrical" because the algorithm computes repeatedly a sequence of geometric intersections between polygons and planes. This idea was optimized until obtaining an algorithm able to run in $O(n)$ operations, which has been proven to be optimal. One natural way to define a method for the 3D kernel computation is to extend the 2D algorithm, which is well studied and documented, to the upper dimension.

The problem with the 3D case is that whereas two convex polygons with respectively n_1 and n_2 vertices can be intersected in time $O(n)$, being $n = n_1 + n_2$, two convex polyhedra with the same parameters are intersected in time $O(n \log n)$, thus the generalization of the two-dimensional instance would yield an $O(n \log^2 n)$ algorithm. This is in contrast with the result shown in [PS85], where a lower bound for the intersection of convex polyhedra is established at $O(n \log n)$.

This brought to the definition of a new algorithm based on the so-called "double duality trick", which makes use of linear algebra and homogeneous coordinates, able to compute the intersection of n half-spaces in time $O(n \log n)$ [PS85]. This algebraic approach can be implemented inside the framework of the CGAL library, although there is currently not an explicit routine for computing the kernel of a polyhedron and one has to connect the function for the intersection of half-spaces to the polyhedron data structure.

While from a theoretical point of view the cited results are indubitable, we believe that in many practical situations the geometric approach could perform better than the algebraic one. Intuitively, if the number of faces and vertices of the polyhedron is low, this method can be more efficient than the algebraic one, as the cost of solving a linear problem does not go under a certain bound while

the intersection of half-spaces can become extremely cheap if performed intelligently.

3. Computing the kernel of a Polyhedron

In this section, we induce our method for computing the kernel of a polyhedron. It has a modular structure composed of four nested algorithms, each one calling the next one in its core part. It is modular in the sense that each algorithm can be entirely replaced by another one performing the same operation(s). For instance, one could use another strategy for computing the intersection between a polygon and a plane and simply replace Algorithm 3 (and Algorithm 4 if not needed).

In the next subsections we adopt the following data structure inherited by the *cinolib* library [Liv19], in which the code has been written:

- *Polyhedron:* class composed by a field *verts* containing the vertices (in 3D coordinates) and a field *faces* containing the faces of a polyhedron.
- *Points:* array of unordered 3D points.
- *Face:* array of unsigned integers representing the indices of the vertices of a face, ordered counter-clockwise.
- *Plane:* class composed by a 3D point *d* indicating a random point on the plane, and a 3D point *n* indicating the unit normal of the plane.

We point out that we always consider a plane *p* together with the direction indicated by its normal vector *p.n*, which is equivalent to considering the half-space originating in *p* and containing *p.n*.

3.1. Polyhedron Kernel

Algorithm 1 tackles the main problem: given a polyhedron *P*, we want to find the polyhedron *K* representing the kernel of *P*. We will also need as input an array containing the outwards normals of the polyhedron faces, as it is not always possible to compute the orientation of a face only from its vertices (for example with non-convex faces).

We start by defining *K* as the axis aligned bounding box (AABB) of *P*, i.e. the box with the smallest volume within which all the vertices of *P* lie, aligned with the axes of the coordinate system. We then recursively "cut" this box with a number of planes. For each face *f* of *P* we compute the plane *p* which contains it, with the orientation given by the opposite of its normal $N(f)$ (that is to say, $p.n = -N(f)$). In general *p* will separate *K* into two polyhedra, and between those two we choose the one containing the vector *p.n*, which points towards the interior of *K*. This operation is performed by the *Polyhedron-Plane-Intersection* algorithm detailed in Section 3.3, which replaces *K* with the chosen polyhedron.

We point out that cutting a convex polyhedron with a plane will always generate two convex polyhedra, and since we start from the bounding box (which is convex), we are guaranteed for *K* to be always a convex polyhedron. We could as well start with considering the polyhedron's convex hull instead of its bounding box, but it would be less efficient because the convex hull costs in general $O(n \log n)$ while the AABB is $O(n)$. Note that even if we used

the convex hull, we would still need to intersect the polyhedron with each of its faces. As we iterate through the faces, we generate a sequence of convex polyhedra K_i, $i = 1, \dots, \#faces$, such that $K_i \subseteq K_{i-1}$. No matter how weird the initial element *P* is, from this point on we will only be dealing with convex polyhedra and convex faces.

Algorithm 1 Polyhedron-Kernel

Input: Polyhedron *P*, Points *N* (faces normals);
Output: Polyhedron *K*
1: $K :=$ AABB of *P*;
2: **for** Face *f* in *P.faces* **do**
3:     Plane $p :=$ plane containing *f* with normal $-N(f)$;
4:     $K :=$ Polyhedron-Plane-Intersection(K, p);
5: **end for**
6: **return** *K*;

3.2. Polyhedron-Plane-Intersection

With the second algorithm we want to intersect a polyhedron *P* with a plane *p*. This intersection will in general be composed of two polyhedra, and between these two we are interested in the one containing the normal vector of *p* (conventionally called the one "above" the plane and indicated with *A*). This algorithm is inspired from [AS08], where the authors define an algorithm for the intersection of a convex polyhedron with an half-space.
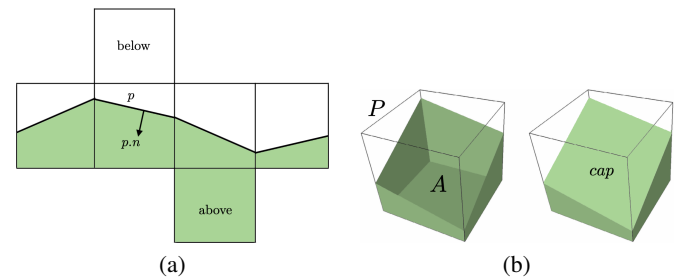

Figure 3: *Intersection of a polyhedron with a plane: (a) clipping and (b) capping of a cube.*

The first part of Algorithm 2 is called the "clipping" part (recalling the terminology from [AS08]) and consists in cutting each face of *P* with the plane *p*, see Fig. 3(a). We preliminarily evaluate the signed distances (defined in Section 2) between the vertices of *P* and the plane to understand their position with respect to *p*. Faces strictly below *p* are discarded, faces weakly above *p* are added to *A* together with their vertices, and faces intersected by *p* are split by the *Polygon-Plane-Intersection* algorithm. While we visit every face only once, the same does not hold for vertices, hence we need to check if a vertex is already present in *A* before adding it.

This simple idea of processing in advance the faces which are only weakly above the plane resolves several implementation issues and in some cases significantly improves the efficiency of the algorithm. By doing this, we make sure that only the faces properly intersected by the plane are passed to Algorithm 3, so that we do not need to implement all the particular cases of intersections in a

single point or along an edge or of faces contained in the plane. In addition, for every face not passed to Algorithm 3 we have an efficiency improvement, and this happens frequently in models with many coplanar faces like the ones considered in Section 4.2.

If at the end of this step p intersects A in more than two points, given that A and all its faces are convex, the vertices shared by p and A will define a "cap" face of A completely contained in p, see Fig. 3(b). In order to sort counter-clockwise the points contained in *capV* we project them onto a plane and apply the algorithm proposed in [Bae21] for 2D points. Note that if the cap face was not convex it would make no sense to order its vertices, but the intersection between a plane and a convex polyhedron will always generate convex faces. Last, we need to check that this face is not already present in A: for example if p is tangent to P along a face, this face could be added to A both as an intersection face and as a cap face. If this is not the case we add *capF* to $A.faces$, but we do not need to add any vertex from *capV*, as we can assume they are all already present in $A.verts$.

Algorithm 2 Polyhedron-Plane-Intersection

Input: Polyhedron P, Plane p
Output: Polyhedron A
1: evaluate the position of all points in *P.verts* with respect to p;
2: **for** Face f in *P.faces* **do**
3:      Points f_v = vertices in *P.verts* relative to f;
4:      **if** all vertices in f_v are strictly below p **then** continue;
5:      **else if** all vertices in f_v are weakly above p **then**
6:          $A.verts \leftarrow f_v$, $A.faces \leftarrow f$;
7:      **else**
8:          $aboveV, aboveF$:=Polygon-Plane-Intersection(f_v, f, p);
9:          $A.verts \leftarrow aboveV$, $A.faces \leftarrow aboveF$;
10:      **end if**
11: **end for**
12: Points $capV$:= vertices in $A.verts$ which are on p;
13: **if** size$(capV) < 3$ **then** return A;
14: **end if**
15: Face $capF$:= $capV$ indices ordered counter-clockwise;
16: **if** $capF \notin A.faces$ **then** $A.faces \leftarrow capF$;
17: **end if**
18: **return** A;

3.3. Polygon-Plane-Intersection

Algorithm 3 describes the intersection of a polygon (representing a face of the polyhedron), defined by an array of 3D points *polyV* and an array of indices *polyF*, with a plane p. In analogy to Algorithm 2, the intersection will in general produce two polygons and we are only interested in the one above the plane, see Fig. 4(a), defined by vertices *aboveV* and indexes *aboveF*. We generically say that a vertex v is added to *above* meaning that v is added to *aboveV* and its index id_v is added to *aboveF*.

We preliminarily evaluate the signed distances between the vertices of the polygon and the plane to understand their position with respect to p. Then we iterate on the edges of *poly*: in order to avoid duplicates, for each couple of consecutive vertices v_1, v_2, we only

accept to add to *above* the second vertex v_2 or the intersection vertex v, but never v_1. When applying the algorithm recursively, as in the case of kernel computation, this idea requires to have all faces oriented coherently.

If both vertices are strictly below p we ignore them, unless v_2 lies exactly on the plane, in which case we add it to *above*. If they are both above or on p we add v_2 to *above*, otherwise we perform the *Line-Plane-Intersection* algorithm and find a new vertex v. Its index id_v will be equal to the maximum value in *polyF* plus one, just to make sure that we are not using the index of an existing vertex. Now, if v_1 is above p (and consequently v_2 is below) we only add v to *above*, while if v_1 is below p (and v_2 is above) we add both v and v_2. As already noted in Section 3.2, treating separately the weak intersections makes the code simpler and more efficient.
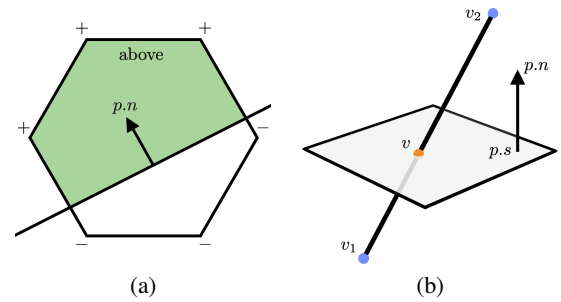

Figure 4: *(a) Intersection between a polygon and a plane, with the above part coloured in green. (b) Intersection between a line and a plane.*

Algorithm 3 Polygon-Plane-Intersection

Input: Points *polyV*, Face *polyF*, Plane p.
Output: Points *aboveV*, Face *aboveF*.
1: evaluate the position of all points in *polyV* with respect to p;
2: **for** $i = 1 : \text{size}(polyF)$ **do**
3:      $id_1 := polyF(i)$, $id_2 := polyF(i+1)$;
4:      $v_1 := polyV(id_1)$, $v_2 := polyV(id_2)$;
5:      **if** v_1 and v_2 are strictly below p **then**
6:          continue;
7:      **else if** v_1 is weakly below p and v_2 is on p **then**
8:          $aboveV \leftarrow v_2$, $aboveF \leftarrow id_2$;
9:      **else if** v_1 and v_2 are weakly above p **then**
10:          $aboveV \leftarrow v_2$, $aboveF \leftarrow id_2$;
11:      **else**
12:          $v :=$ Line-Plane-Intersection(v_1, v_2, p);
13:          $id_v := \max(polyF)+1$;
14:          **if** v_1 is strictly above p **then**
15:             $aboveV \leftarrow v$, $aboveF \leftarrow id_v$;
16:          **else**
17:             $aboveV \leftarrow v$, $aboveF \leftarrow id_v$;
18:             $aboveV \leftarrow v_2$, $aboveF \leftarrow id_2$;
19:          **end if**
20:      **end if**
21: **end for**
22: **return** *aboveV*, *aboveF*;

3.4. Line-Plane-Intersection

This last algorithm returns the intersection point between a line, given as a couple of vertices, and a plane. It is a very simple and well known procedure, and we report it here only for completeness.

The intersection vertex v is defined by a linear combination of v_1 and v_2, with a coefficient t which may also fall outside the standard range $[0,1]$. The only difficulty lies in the definition of t, which is the negative ratio between two scalar products involving the plane normal $p.n$ and a generic other point on the plane, $p.s$, other than v_1 and v_2, see Fig 4(b). If the denominator D vanishes, it means that either the line does not intersect the plane or that the line is contained in it: we treat these exceptions as errors because in Algorithm 3 we only call this algorithm after checking that the edge (v_1, v_2) properly intersects the plane p.

Algorithm 4 Line-Plane-Intersection

Input: vertices v_1, v_2, Plane p.
Output: vertex v.
1: $N := (p.n) \cdot (v_1 - p.s)$;
2: $D := (p.n) \cdot (v_2 - v_1)$;
3: **if** $D = 0$ & $N \neq 0$ **then** error: no intersections;
4: **else if** $D = 0$ & $N = 0$ **then** error: line contained in the plane;
5: **end if**
6: $t := -N/D$;
7: **return** $v := v_1 + t(v_2 - v_1)$;

3.5. Computational complexity

Making advantage of the modularity of our algorithms, we can estimate separately the computational cost of each algorithm and then include them into a single formula.

We start with a polyhedron P with n_v vertices and n_f faces. For Algorithm 1 we need to compute its AABB, which is $O(n_v)$, and then perform Algorithm 2 for n_f times, therefore $N_1 = n_v + n_f N_2$.

In Algorithm 2 we receive as input a polyhedron which is potentially different from P, but we empirically measured that the number of vertices and faces remain approximately constant when applying this routine. We preliminarily compute n_v signed distances; then, only in the cases in which the vertices of a face do not all lie by the same side, run Algorithm 3. If only m_1 faces needs to be cut (with $m_1 < n_f$), the clipping part takes $n_v + m_1 N_3$ operations. If we have a cap face (which is not always true) with n_{vc} verts, we need to sort them with a QuickSort routine which is on average $O(n_{vc} \log n_{vc})$. Altogether, Algorithm 2 takes $N_2 = n_v + m_1 N_3 + n_{vc} \log n_{vc}$ operations.

Given a face f_i with n_{vi} vertices, in Algorithm 3 we start by computing n_{vi} signed distances. Then, if only m_2 of the edges intersect the plane (where $m_2 < n_{vi}$), perform Algorithm 4. This means that $N_3 = n_{vi} + m_2 N_4$, and Algorithm 4 only consists of 4 operations.

Collecting all costs together, we obtain:

$$N_4 = 4;$$
$$N_3 = n_{vi} + 4m_2;$$
$$N_2 = n_v + m_1 (n_{vi} + 4m_2) + n_{vc} \log n_{vc};$$
$$N_1 = n_v + n_f (n_v + m_1 (n_{vi} + 4m_2) + n_{vc} \log n_{vc});$$
$$= n_v + n_f (n_v + C).$$

Let us focus our attention on the term C. Since $m_2 \leq n_{vi}$ and n_{vc} is the number of vertices of the cap face, we can substitute them with an average n_a of the number of vertices per face and get $C = m_1 n_a + n_a \log n_a$. For the very majority of the considered models (especially the more complex ones) we can assume that both m_1 and n_a are negligible compared to n_v: a plane can intersect a polyhedron in a very limited number of its faces, and the average number of vertices per face is significantly smaller than the total number of vertices. The term C can therefore be included in n_v, and as a realistic approximation of the computational cost we get $O(n_v(1 + n_f))$.

4. Examples and discussions

In this section we test our method over a collection of polyhedra, comparing its performance to the results obtained using our implementation of the algebraic method in CGAL. Experiments have been performed on a MacBook Pro equipped with a 2,3 GHz Intel Core i5 processor with four CPUs and 16GB of RAM. Source code is written in C++ and it is accessible at <https://github.com/TommasoSorgente/polyhedron_kernel> together with all datasets.

4.1. Collections of polyhedral elements

As an initial test we built four datasets containing 1000 non-convex polyhedra each, simulating a typical configuration in which a tessellation of a complex 3D computational domain come with non-convex elements. As elements of a tessellation on which to perform numerical simulations, they typically present a fairly simple shape and possess a fairly limited number of vertices. A gallery of examples is visible in Fig. 5.

The first three are denoted *tet10*, *tet20* and *tet30*, where the number indicates the number of vertices of the contained polyhedra. Each polyhedron is built by sampling the relative number of points randomly in the space and connecting them in a Delaunay tetrahedalization using *Tetgen* [Si15]. Then one vertex is moved to the polyhedron's centroid, in order to get a non convex element. Note that *Tetgen* outputs the convex hull of the Delaunay tetrahedralization of a set of points, therefore the initial polyhedron is guaranteed to be convex, its centroid is guaranteed to lie inside its volume and the operation of moving a vertex towards the polyhedron centroid does not generate connectivity problems.

The fourth dataset, *voro*, contains 1000 random non-convex polyhedra with non-triangular faces. For building each polyhedron we start from a Voronoi cell and triangulate the face with the largest area connecting the face vertices to the face centroid. Then we move the face centroid to the polyhedron centroid to obtain a non
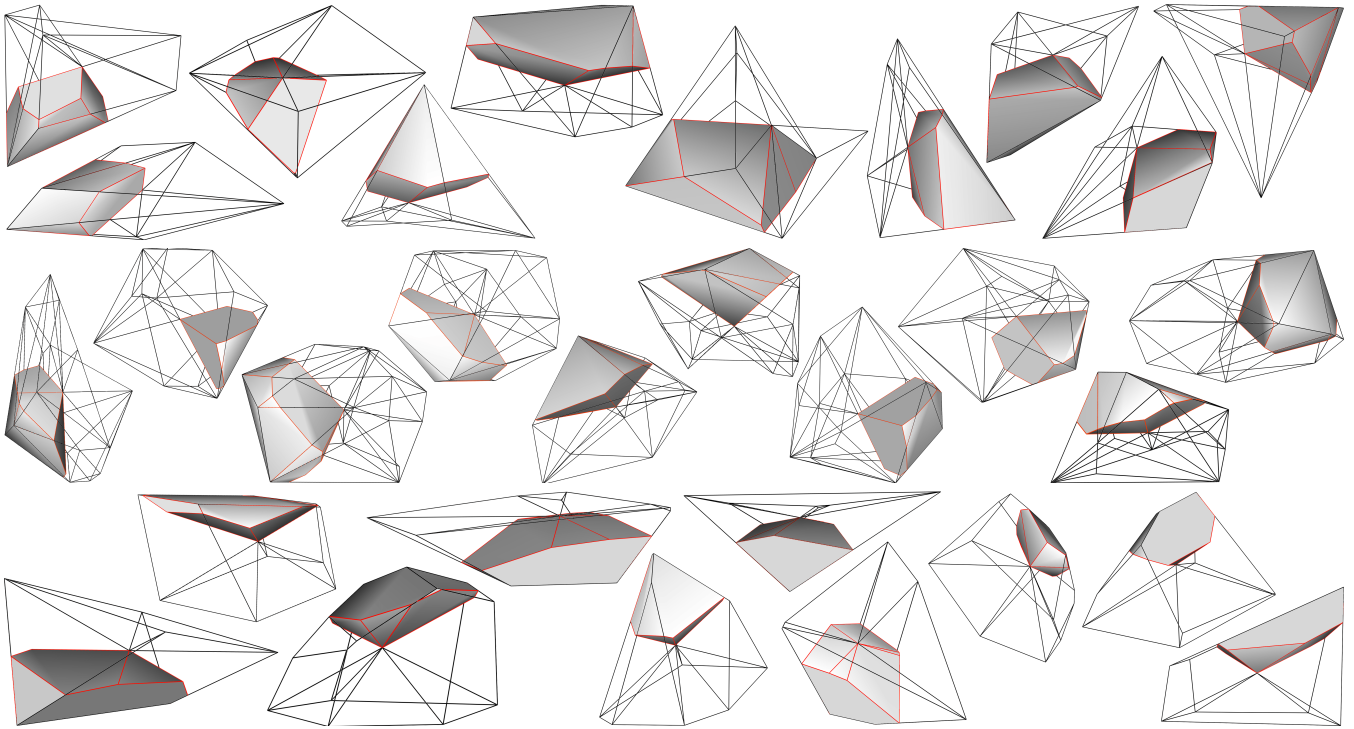
Figure 5: *Polyhedra and kernels from datasets tet10, tet30 and voro.*

convex element. It is not possible to know a priori the exact number of vertices of each Voronoi cell, but after the definition of dataset *voro* we measured that the number of vertices of its elements ranges from 5 to 20.

Table 1: *Computational times for estimating the kernel on the whole dataset (in seconds) and ratio between the CGAL time and ours.*

dataset	#vertices	our	CGAL	ratio
tet10	10	0.29	2.49	8.58
tet20	20	0.49	3.62	7.39
tet30	30	0.63	4.49	7.13
voro	5-20	0.24	1.92	8

We measure the computational time needed to estimate the kernels of all polyhedra in each dataset and compare it with the performance of the CGAL library in Table 1. On this kind of polyhedra, the geometrical approach performs significantly better (between seven and eight times) than the algebraic one. This may be due to the fact that, with a limited number of planes, computing the geometric intersection between them is cheaper than solving a linear problem. As the number of vertices increases the difference between the two approaches becomes smaller.

4.2. Refinements

As a second setting for our tests we wanted to measure the asymptotic behaviour of our method as the number of vertices increases. We considered two polyhedra taken from the dataset *Thingi10K* [ZJ16]: the so-called *laser-chess* and *flex*. These models are given in the form of a surface mesh and we treat them as single volumetric cell, analyzing the performance of our algorithm as we refine them.
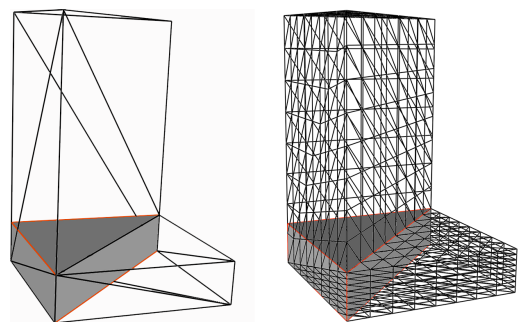

Figure 6: *Original laser-chess model and its third refinement. The kernels are identical.*

The *laser-chess* model is extremely simple, being defined by only eight planes. Even refining its faces with a midpoint strategy, the planes induced by its faces remain identical and the kernels of the refined models are all equal (Fig. 6). On this example

our method performs on average 6.8 times better that the algebraic method (see Table 2), and the computational time scales with a constant rate (see the red lines in Fig. 8). Our implementation takes advantage of the fact that Algorithm 2 recognises coplanar faces and performs Algorithm 3 only eight times, independently of the number of faces.
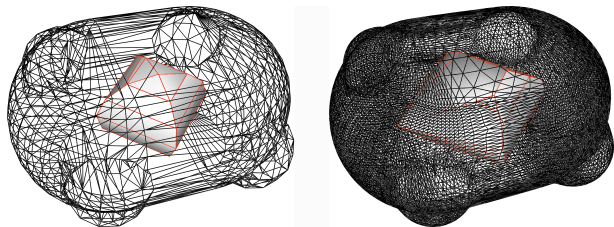

Figure 7: *Original flex model and its third refinement. Small perturbations in the faces lead to slightly different kernels.*

The *flex* model is more complex, as it presents a curved surface which generates a lot of different planes defining the kernel. Moreover, we refined this model using the Loop's algorithm and this generated faces lying on completely new planes. This explains the difference between the two kernels in Fig. 7. Our geometric method improves the performance of the algebraic one by one order of magnitude, even if the difference between the two approaches decreases (from 13 times to 9, see Table. 2) as the number of vertices increases.

Table 2: *Computational times for the laser-chess and flex refinements (in seconds) and ratio between the CGAL time and ours.*

mesh	#vertices	our	CGAL	ratio
laser-chess1	10	0	0.001	1
laser-chess2	127	0.001	0.007	7
laser-chess3	493	0.004	0.027	6.75
laser-chess4	1945	0.017	0.11	6.47
laser-chess5	7729	0.06	0.41	6.83
laser-chess6	30817	0.24	1.73	7.21
flex1	834	0.026	0.35	13.46
flex2	3130	0.097	1.17	12.06
flex3	11216	0.45	5.1	11.33
flex4	26560	1.33	14.19	10.67
flex5	35566	2.31	22.1	9.57
flex6	42659	3.57	33.26	9.32

4.3. Complex models

Last, we try to compute the kernel of some more complex models, taken again from the dataset *Thingi10K* and treated as single volumetric cells. Note that most of the models in Thingi10K are not star-shaped, thus making it useless to compute their kernels. Even if our method is designed for dealing with a large number of simple polyhedra (i.e., with a relatively small number of vertices), our algorithms are still able to compute the kernel of a whole object with thousands of vertices. In Fig. 9 we present the kernel computation
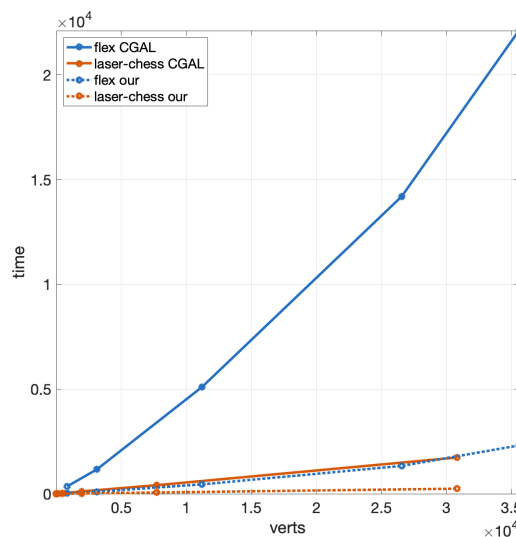

Figure 8: *Computational times for the refinement of the flex and laser-chess models. Time is expressed in milliseconds.*

of six complex objects, and the computational times are reported in Table 3.

Table 3: *Computational times for complex models (in seconds) and ratio between the CGAL time and ours.*

mesh	#vertices	our	CGAL	ratio
bot-eye	453	0.28	0.048	0.17
rt4-arm	655	0.21	0.11	0.52
super-ellipse	290	0.02	0.04	2
playset	1416	0.01	0.08	8
leg	87	0.003	0.03	10
fandisk	7229	0.07	3.58	51.14

We notice that the number of vertices of the element, by itself, is not strictly related to the efficiency of the method. Much more influence has the shape of the object: in accordance to the results of Section 4.2, over simple models like *leg* or models with numerous ajacent coplanar faces like *playset* and *fandisk* our method is preferable. Note in particular the different performance of the two methods over the *fandisk* model, which has a high number of vertices. Vice versa, over elements with significant curvatures like *bot-eye*, *rt4-arm* or *super-ellipse*, the algebraic method performs similarly or better than ours even on relatively small models.

5. Conclusions

We presented an algorithm for the computation of the kernel of a polyhedron based on the extension to the 3D case of the geometric approach adopted in two dimensions. The algorithm showed up to be robust and reliable, as it computed successfully the kernel of every considered polyhedron.

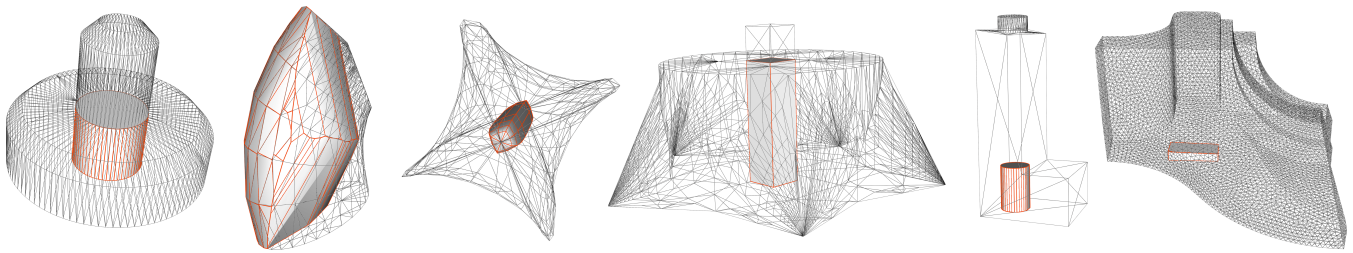We compared its efficiency to the one of the algebraic approach

Figure 9: *Examples of our kernel evaluation for complex models: bot-eye, rt4-arm, super-ellipse, playset, leg, fandisk.*

to the problem, implemented in CGAL. From a theoretical point of view, the computational complexity evaluation of Section 3.5 suggests that our method is in general quadratic, while the algebraic approach has a lower bound at $n \log(n)$. Nonetheless, we proved in Section 4 that in several circumstances our approach outperforms the algebraic one.

Our method performs significantly better than the algebraic approach over polyhedra with a limited number of vertices and faces, as shown in Section 4.1, making it particularly suitable for the analysis of volumetric tessellations with non-convex elements. Indeed, we point out that our algorithm is specifically designed to be used with simple polyhedra, possibly composing a bigger and more complex 3D model, and not with a complete model itself.

When the size of the polyhedron increases, our method is still particularly efficient if the model has numerous coplanar faces, like in some of the complex examples in Section 4.3. This is a very common situation in models representing mechanical parts. On the other side the algebraic approach is preferable over curved domains, with numerous vertices and faces lying on different planes.

In conclusion, with this work we do not aim at completely replacing the algebraic approach for the kernel computation but instead to give an alternative which can be preferred for specific cases, such as the quality analysis of the elements in a 3D tessellation, in the same way as bubble-sort is to be preferred to optimal sorting algorithms when dealing with very small arrays.

As a future development, we plan to include this tool in a suite for the generation and analysis of tessellations of three dimensional domains, aimed at PDE simulations. The kernel of a polyhedron has a great impact on its geometrical quality, and the geometrical quality of the elements of a mesh determines the accuracy and the efficiency of a numerical method over it. We therefore plan to use this algorithm for better understanding the correlations between the shape of the elements and the performance of the numerical simulations, and be able to adaptively generate, refine or fix a tessellation accordingly to them.

Acknowledgements

We would like to thank Dr. M. Manzini for the precious discussions and suggestions, and all the people from IMATI institute involved in the CHANGE project. Special thanks are also given to the anonymous reviewers for their comments and suggestions.

References

[ABB*21] ATTENE, MARCO, BIASOTTI, SILVIA, BERTOLUZZA, SILVIA, et al. "Benchmarking the geometrical robustness of a Virtual Element Poisson solver". *Mathematics and Computers in Simulation* 190 (2021), 1392–1414. ISSN: 0378-4754. DOI: https://doi.org/10.1016/j.matcom.2021.07.018 2.

[AS08] AHN, HYUNG T. and SHASHKOV, MIKHAIL. "Geometric algorithms for 3D interface reconstruction". *Proceedings of the 16th international meshing roundtable*. Springer. 2008, 405–422 3.

[Bae21] BAELDUNG. *Baeldung guides and courses*. https://www.baeldung.com/cs/sort-points-clockwise. Last accessed 2021-09-08. 2021 4.

[BBC*13] BEIRÃO DA VEIGA, LAURENÇO, BREZZI, FRANCO, CANGIANI, ANDREA, et al. "Basic principles of virtual element methods". *Mathematical Models and Methods in Applied Sciences* 23.01 (2013), 199–214 2.

[Boo21] BOOST. *Boost C++ Libraries*. http://www.boost.org/. Last accessed 2021-09-08. 2021 1.

[Cia02] CIARLET, PHILIPPE G. *The finite element method for elliptic problems*. SIAM, 2002 1.

[CKS12] COCKBURN, BERNARDO, KARNIADAKIS, GEORGE E., and SHU, CHI-WANG. *Discontinuous Galerkin methods: theory, computation and applications*. Vol. 11. Springer Science & Business Media, 2012 2.

[DAB18] DEMIR, İLKE, ALIAGA, DANIEL G, and BENES, BEDRICH. "Near-convex decomposition and layering for efficient 3D printing". *Additive manufacturing* 21 (2018), 383–394 1.

[DD19] DI PIETRO, DANIELE A. and DRONIOU, JÉRÔME. *The Hybrid High-Order method for polytopal meshes*. Vol. 19. Springer, 2019 2.

[DS80] DUPONT, TODD and SCOTT, RIDGWAY. "Polynomial approximation of functions in Sobolev spaces". *Mathematics of Computation* 34.150 (1980), 441–463 2.

[FP09] FABRI, ANDREAS and PION, SYLVAIN. "CGAL: The computational geometry algorithms library". *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*. 2009, 538–539 1.

[JP17] JACOBSON, ALEC and PANOZZO, DANIELE. "Libigl: prototyping geometry processing research in C++". *SIGGRAPH Asia 2017 courses*. 2017, 1–172 1.

[LF15] LÉVY, BRUNO and FILBOIS, ALAIN. "Geogram: a library for geometric algorithms". (2015) 1.

[Liv19] LIVESU, MARCO. "cinolib: a generic programming header only C++ library for processing polygonal and polyhedral meshes". *Transactions on Computational Science XXXIV*. Springer, 2019, 64–76 3.

[LMS14] LIPNIKOV, KONSTANTIN, MANZINI, GIANMARCO, and SHASHKOV, MIKHAIL. "Mimetic finite difference method". *Journal of Computational Physics* 257 (2014), 1163–1227 2.

[LP79] LEE, DER-TSAI and PREPARATA, FRANCO P. "An Optimal Algorithm for Finding the Kernel of a Polygon". *J. ACM* 26.3 (July 1979), 415–421. ISSN: 0004-5411. DOI: 10.1145/322139.322142 1.

[PS85] PREPARATA, FRANCO P. and SHAMOS, MICHAEL I. *Computational Geometry: An Introduction*. Berlin, Heidelberg: Springer-Verlag, 1985. ISBN: 0387961313 1, 2.

[SB08] SCOTT, RIDGWAY and BRENNER, SUSANNE C. *The mathematical theory of finite element methods*. 3rd ed. Texts in applied mathematics 15. Springer-Verlag New York, 2008. URL: http://gen.lib.rus.ec/book/index.php?md5=9ad140b0da07318a38c12d35e5616cc2 2.

[SBMS21] SORGENTE, TOMMASO, BIASOTTI, SILVIA, MANZINI, GIANMARCO, and SPAGNUOLO, MICHELA. "The role of mesh quality and mesh quality indicators in the Virtual Element Method". *arXiv preprint arXiv:2102.04138* (2021) 2.

[SH76] SHAMOS, MICHAEL I. and HOEY, DAN. "Geometric intersection problems". *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. 1976, 208–215. DOI: 10.1109/SFCS.1976.16 1.

[Si15] SI, HANG. "TetGen, a Delaunay-based quality tetrahedral mesh generator". *ACM Transactions on Mathematical Software (TOMS)* 41.2 (2015), 1–36 5.

[SPC*21] SORGENTE, TOMMASO, PRADA, DANIELE, CABIDDU, DANIELA, et al. "VEM and the Mesh". *arXiv preprint arXiv:2103.01614* (2021) 2.

[ZJ16] ZHOU, QINGNAN and JACOBSON, ALEC. *Thingi10K: A Dataset of 10,000 3D-Printing Models*. 2016. arXiv: 1605.04797 [cs.GR] 6.