Eurographics Symposium on Geometry Processing 2025
M. Attene and S. Sellán
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 44 (2025), Number 5

Exact and Efficient Mesh-Kernel Generation

J. Nehring-Wirxel[1] and P. Kern[1] and P. Trettner[2] and L. Kobbelt[1]

[1]RWTH Aachen University, Germany
[2]Shaped Code GmbH, Germany

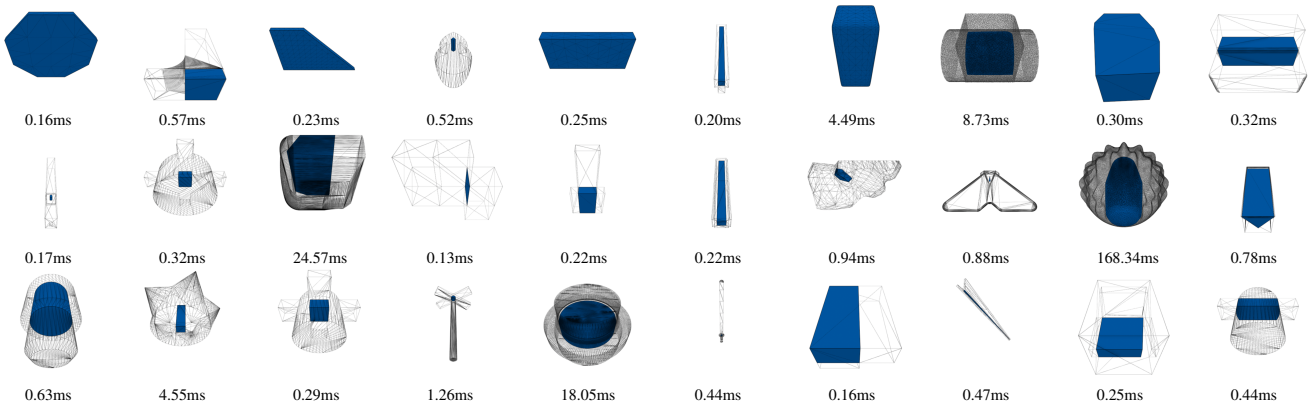


Figure 1: *Our approach efficiently and exactly computes the kernel of a triangle mesh by iteratively intersecting the half-spaces defined by the input triangles. The kernel is visualized as the convex blue polyhedron on the inside of each of the shown meshes. The computation time per kernel and mesh is given in ms.*

Abstract

The mesh kernel for a star-shaped mesh is a convex polyhedron given by the intersection of all half-spaces defined by the faces of the input mesh. For all non-star-shaped meshes, the kernel is empty. We present a method to robustly and efficiently compute the kernel of an input triangle mesh by using exact plane-based integer arithmetic to compute the mesh kernel. We make use of several ways to accelerate the computation time. Since many applications just require information if a non-empty mesh kernel exists, we also propose a method to efficiently determine whether a kernel exists by developing an exact plane-based linear program solver. We evaluate our method on a large dataset of triangle meshes and show that in contrast to previous methods, our approach is exact and robust while maintaining a high performance. It is on average two orders of magnitude faster than other exact state-of-the-art methods and often about one order of magnitude faster than non-exact methods.

CCS Concepts

• ***Computing methodologies*** → *Mesh geometry models;* • ***Theory of computation*** → *Linear programming;* • ***Applied computing*** → *Computer-aided design;*

1. Introduction

The kernel of a 3D mesh is the set of all points from which every point on the mesh surface is visible. This means that any point that is part of the kernel can be connected to any point of the mesh surface by a straight line that does not intersect the mesh. The class of meshes that have a non-empty kernel is restricted to star-shaped meshes, and for many applications a check for the existence of a kernel is sufficient.

Applications include finding scaling centers for *scaled boundary* finite element methods [Wol03], constructing view regions for autonomous exploration [XZWC24], and star-shaped decomposition [YL11].

While the efficient computation of kernels is a long-standing problem, there have been some recent developments for computing the kernel of a 3D polyhedron [AS24a; SBS22], but their available implemenations do not use exact arithmetics and can therefore output incorrect results. For downstream applications, it is crucial that

the outcome of the kernel computation is correct while maintaining computational efficiency.

In this paper, we present an exact and efficient method for computing the kernel of a 3D mesh. It is both faster than previous non-exact methods and is guaranteed to produce correct results. We make use of a plane-based representation of the mesh, which allows us to robustly compute the kernel. We propose to use a set of methods to accelerate the intersection computation. Additionally, we propose a purely plane-based exact linear problem solver to quickly determine the existence of a kernel, and generate a point in the kernel, if it exists.

2. Related Work

Kernel Computation

Early work on kernel computation was conducted by Shamos and Hoey [SH76], who establish that the optimal time complexity for intersecting h half-spaces is $O(h \log h)$. They also observe that this lower bound does not extend to kernel computation, as the edge order is not arbitrary. Building on this, Lee et al. [LP79] propose an $O(e)$-time algorithm for computing the kernel of a 2D polygon, where e denotes the number of edges. Many modern computational geometry libraries, such as CGAL [The24], implement the $O(h \log h)$ algorithm by Preparata et al. [PM79]. This method computes the convex hull of the dual points and projecting the result back to the primal space. Kernel computation has received increased attention in recent years. Despite the good theoretical complexity guarantees, the algorithm is slow in practice and current implementations often fail to compute degenerate kernels. Sorgente et al. [SBS22] propose a geometric approach similar to ours, computing the kernel by intersecting the polyhedron's half-spaces. However, their method scales poorly for more complex kernel meshes. To improve upon this, Asiler et al. [AS24b] introduce an $O(f \cdot r)$-time algorithm for approximating the kernel of a 3D polyhedron using sampling rays, where f and r denote the number of faces and rays, respectively. Additionally, they introduce a method that first identifies an initial kernel vertex and subsequently traces the kernel's edges via plane intersections [AS24a]. Unfortunately, the implementations for both [SBS22] and [AS24b] suffer from numerical robustness issues and can produce incorrect results. Both methods can likely replace inexact floating point computations with exact arithmetics or predicates to improve stability, but this usually comes with a high computational price, especially since both methods use the square root for vector normalization.

Kernel Usage

Kernel computation is a fundamental operation in computational geometry with applications across various fields. Livesu [Liv24] presents a method that embeds a mesh domain, which must however have a non-empty kernel. Kernel computation also plays a crucial role in star-shaped decomposition, where some algorithms rely heavily on its computation. Yu et al. [YL11] propose a method for star-shaped decomposition by solving an integer linear program to determine guard positions, i.e. points inside a kernel, followed by constraint region growing to extract the corresponding star-shaped regions. Hinderink et al. [HC23] explore the construction of a continuous bijection between a mesh and a star-shaped domain, which benefits from efficient kernel computation. They later integrate bijection mapping with star-shaped decomposition, demonstrating that arbitrary ball-topology models can be decomposed into multiple star-shaped region mappings [HBC24]. In finite element methods, kernel computation is used to assess mesh quality, measuring shape regularity and star-shapedness [SBMS22]. In autonomous exploration, Xu et al. [XZWC24] apply kernel computation to construct view regions.

Precision and Exactness in Linear Programming

There has been significant effort in computing exact solutions to linear programs. Dhiflaoui et al. [DFK*03] combine rational arithmetic with the simplex method to develop an algorithm that verifies the feasibility of a basis for a primal or dual linear program. Their method can also correct the given solutions or compute new ones. Since it uses the simplex method, it scales poorly for problems containing many constraints. Koch [Koc03] benchmarks solutions to the NETLIB instances, refining floating-point precision where existing algorithms failed. Applegate et al. [ACDE07] follow a similar approach. They first compute an approximate solution and then verify it using rational arithmetic. If the solution is incorrect, they increase the precision and iterate the process which requires more computation time.

Plane-Based Representations

The usage of planes as basic primitives to represent other geometric entities is well established as an alternative to vertex-based representations. The general advantage is numerical robustness, as it can be avoided to compute the intersections of the primitives directly, introducing a rounding error in the process. An important use-case is in mesh Booleans, where robust intersections are crucial. Bernstein et al. [BF09] propose a method to robustly perform Booleans where the Geometry is represented by binary space partitioning trees (BSP). They use exact floating point predicates similar to [Ric97], or more recently [Att20] to classify a plane against points represented as the intersection of three other planes. Since Booleans on BSPs scale poorly, [CK10] expand on this idea by placing individual BSP trees inside an octree to reduce the individual tree complexity. [NTK21] propose a similar method, but instead make use of integer predicates to robustly perform iterated Boolean operations. [TNK22] build upon the integer predicates to efficiently compute mesh Booleans without requiring BSP trees. Our proposed method uses the predicates and representation from [NTK21; TNK22] to efficiently and robustly compute the mesh kernel, as well as evaluate an exact linear program.

Contribution

Our method is based on the idea of computing the kernel by cutting a large enough input cube with the supporting planes of the input mesh from [SBS22]. This is combined with the insight from [AS24a] that the existence of a kernel can be quickly evaluated by first finding a kernel vertex using an LP solver. Both ideas are married with the exact plane-based integer arithmetic from [NTK21;

TNK22] to guarantee algorithmic robustness, and the correctness of the kernel while maintaining good performance. We accelerate the cutting of the kernel geometry with several optimizations such as using a proxy geometry and propose an exact plane-based LP solver to quickly and exactly check for the existence of a mesh kernel. Our combined method is both an order of magnitude faster than previous, even approximate, methods and is guaranteed to produce correct results.

3. Computing the Mesh-Kernel

3.1. Kernel Definition

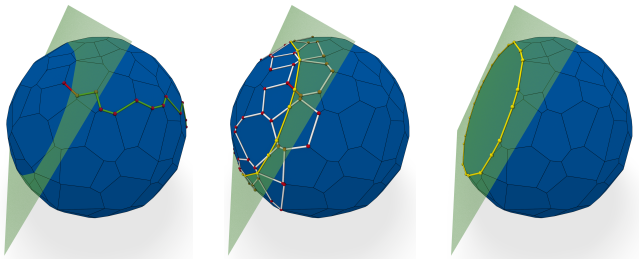
Given a triangle mesh M consisting of k oriented triangles $t_i = (v_a, v_b, v_c), v_j \in \mathbb{R}^3, i < k$ with outward pointing normals $n_i = (v_b - v_a) \times (v_b - v_a)$ and $d_i = -n_i^\mathsf{T} v_a$, the supporting plane p_i of each triangle t_i is given by $n_i^\mathsf{T} x + d_i = 0$. The mesh kernel $\mathcal{K}$ is then the intersection of the negative half-spaces of all supporting planes p_i:

$$\mathcal{K}(M) = \{x \in \mathbb{R}^3 \mid \forall i : \; n_i^\mathsf{T} x + d_i \leq 0\}$$

Since the intersection of convex half-spaces is convex, the mesh kernel must either be a convex polyhedron or empty.

3.2. Overview

Similar to [SBS22], our method computes the mesh kernel by initializing an intermediate kernel polyhedron $\hat{\mathcal{K}}$ with an axis aligned cuboid that contains the entire input geometry, and then iteratively subtracting the half-spaces defined by the input faces from $\hat{\mathcal{K}}$ until either all half-spaces have been subtracted or $\hat{\mathcal{K}}$ is empty as shown in Figure 3. To enable our exact arithmetic framework, the initial input coordinates are scaled up to use at most 26 bits and rounded to integer. This is akin to employing fix-point numbers with 26 bits, which is usually enough to represent common floating point meshes with 23 bits mantissa exactly and produces only a marginal absolute error otherwise.



(a) (b) (c)

Figure 2: *The three steps of the efficient polyhedron cutting method from [NTK21]: (a) Trace greedily from a starting position towards the cutting plane until a point on it or a segment crossing it is found. (b) From the first found intersection, zipper around the adjacent faces and subdivide edges and faces that intersect with the cutting plane. (c) Remove the geometry on the undesired side of the cutting plane and fill the remaining hole with a face.*

3.3. Convex Polyhedron Intersections

Given the current intermediate mesh-kernel $\hat{\mathcal{K}}$, we select the next plane and cut $\hat{\mathcal{K}}$ with it. The geometry on the positive side of the plane is then discarded.

To manage the increasing complexity of $\hat{\mathcal{K}}$ – and therefore the complexity of the mesh cutting – we employ the same strategy explained in detail in Section 3.3 of [NTK21], to quickly cut a convex polyhedron with a plane. A brief overview is shown in Figure 2. Starting at a random vertex on $\hat{\mathcal{K}}$, we greedily walk across the edges of the polyhedron that bring us closer to its intersection with the plane. As soon as we find an edge that intersects with the plane, we *zipper* around the adjacent faces that intersect with the plane and along the way split all edges that intersect with the cutting plane. Each visited face is then split into two faces, one that is entirely on the negative side of the plane, and one that is entirely on the positive side of the plane. Then, we find a starting vertex adjacent to one of the intersection vertices that is on the positive side of the cutting plane and perform a depth-first search to find all vertices that are on the positive side of the cutting plane and remove them. The remaining hole is then filled by a new face that lies entirely inside the cutting plane, resulting in the remaining intermediate mesh-kernel. As shown in [NTK21] this method is significantly faster than classifying all mesh kernel vertices and then performing the cut for all intersected faces.

3.3.1. Degenerate Kernel

It can happen that a cutting plane p_i only touches the surface of $\hat{\mathcal{K}}$. We detect this case if no edge is properly cut by p_i and all vertices exactly on p_i only have neighbors on the same side of p_i. If there are only neighbors on the negative side of p_i, then all of $\hat{\mathcal{K}}$ is still part of the current kernel. Otherwise only the surface that is touched by p_i can be part of the resulting mesh kernel as the remainder lies on the positive side that will be cut away. If only one or two vertices lie exactly on the plane, $\hat{\mathcal{K}}$ now only consists of a single vertex or a line segment connecting the two vertices. If more than two vertices lie exactly on p_i, they must all be from the same convex boundary polygon of $\hat{\mathcal{K}}$. The remaining vertex, line segment, or convex polygon still use the plane-based representation from [NTK21; TNK22] (cf. Section 3.4) and are then clipped against the remaining cutting planes to determine the mesh kernel. Since these kernels are non-empty but contain no volume, we consider them *degenerate*.

3.4. Exact Arithmetic

To assure that our method is both exact and efficient, we make use of the integer plane-based framework presented in [NTK21; TNK22]. In this formulation, all geometric primitives are represented by one or more hyperplanes, where each hyperplane is given by four integer coefficients a, b, c, d that fulfill the plane equation $ax + by + cz + d = 0$ for any point $(x, y, z)^\mathsf{T}$ on the plane. The first three coefficients are a non-normalized normal vector $\vec{n} = (a, b, c)^\mathsf{T}$ and $d = -\vec{n}^\mathsf{T} p$ for a point p on the plane.

Points are represented as the intersection of three non-coplanar hyperplanes, lines are given by the intersection of two hyperplanes, and convex polygons are given by a supporting plane and a list of

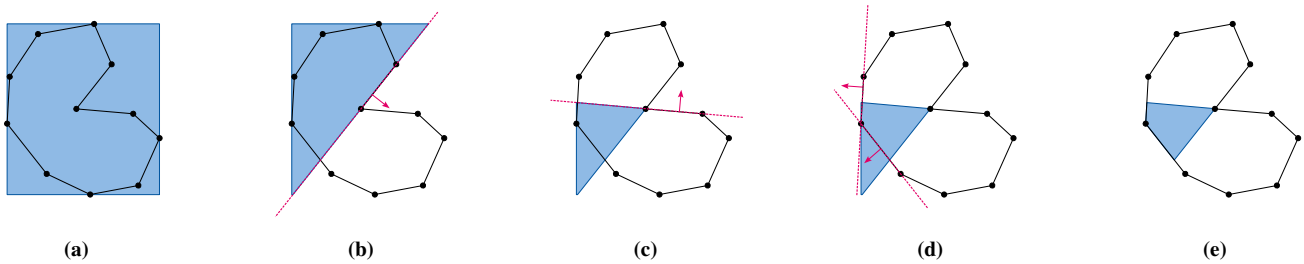


Figure 3: *To compute the mesh kernel, we initialize the intermediate kernel with the axis-aligned bounding box of the input mesh (a). We then iteratively cut the current kernel with the supporting planes of the input mesh (b-d), starting with the supporting planes adjacent to concave edges. The final kernel is either a convex polyhedron (e) or empty. We have omitted the intermediate steps that do not intersect with the current polyhedron kernel for clarity.*

planes that are non-coplanar to the supporting plane, that limit the bounds of the convex polygon. Furthermore, rays are like points also represented as triplet of planes. However, in this case the order matters, as two planes define the line in which the ray lies, and the third plane defines the starting point and direction of the ray. Similarly, segments are represented as two planes that define the line in which the segment lies, and two planes that define the starting and ending point of the segment.

In addition, we use the function $\texttt{intersect}(p,q,r) = (x,y,z,w)^\intercal$ from [NTK21] to compute the exact intersection of three integer planes p,q,r. The result is a point in homogeneous integer coordinates. We also use the $\texttt{classify}(u,(x,y,z,w))$ predicate to check exactly whether the point lies behind, on, or in front plane u. The predicates in [NTK21] guarantee efficient computation with a limited number of intermediate bits, assuming the input integer coordinates defining the planes also have bounded bit-width. For the number of input bits, we choose 26 bits for the input points which guarantees that the most expensive operations can be computed with at most 256-bit integer numbers [NTK21]. Therefore, we scale the input mesh's bounding box to $[\pm 2^{26-1}]^3$ and round to integer coordinates once, before computing the mesh kernel.

3.5. Acceleration

Since the plane-polyhedron cutting takes the largest chunk of our computation time, we employ various strategies to speed up this operation:

1. Terminate early for convex or genus > 0 inputs
2. Unify connected sets of coplanar faces
3. Apply all concave cuts first
4. Proxy the mesh kernel with an axis-aligned bounding box
5. Start intersection tracing at a previously cut vertex

Additionally, we perform a check wether the input even has a non-empty mesh kernel using an exact linear program in a parallel thread and terminate immediately once either method has determined that the kernel is empty. The exact linear program solver is covered in Section 3.6. Let us now cover each of these strategies in more detail:

3.5.1. Early Termination

There are two conditions that allow an early termination before computing the kernel polyhedron or checking for LP feasibility in case that our input is a closed manifold mesh: If the input mesh is convex, it and its kernel are identical, and we can terminate immediately. If the input has a genus other than zero, the mesh kernel is guaranteed to be empty. We can check the genus directly after loading the input by checking the Euler Characteristic.

3.5.2. Coplanar Planes

Many real-world meshes contain a significant portion of coplanar faces. To avoid redundant computations, we collect all connected coplanar faces only once using a union-find, or sometimes called *disjoint-set* data structure [Tar75] that allows the efficient collection of disjoint elements that belong to the same set. This will not find all duplicate planes since there might be multiple disconnected coplanar regions, but will reduce the number of planes that need to be checked for intersection with the current polyhedron. We can also use a hash set to check for duplicate planes. However, since the integer plane representation is not unique by default, i.e. the plane (a,b,c,d) is the same as the plane $(a \cdot w, b \cdot w, c \cdot w, d \cdot w)$ for any integer w, we must first find the greatest common denominator for the coefficients of each plane and divide by it to get a unique representation. We empirically found that the union-find data structure is more efficient in practice (cf. Table 2).

3.5.3. Concave Cuts

In general, it is beneficial to perform as few cuts to the intermediate polyhedron as possible. However, finding a minimal subset of the input cutting planes is difficult.

While convex regions of an input mesh may or may not be part of the resulting mesh kernel, each concave edge is initially guaranteed to remove geometry from the kernel polyhedron, as each concave edge automatically produces a region that is not visible from the other side. Therefore, it is more likely, that planes at concave edges remove larger parts of the current mesh kernel polyhedron. Hence, we perform all cuts with supporting planes adjacent to concave edges first.

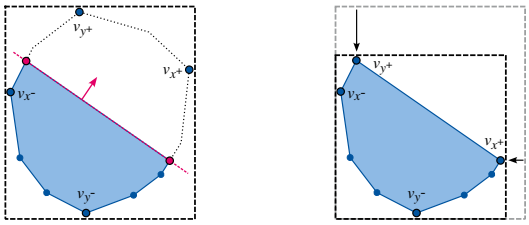


Figure 4: *After cutting the current polyhedron, the AABB is updated if any of the extreme vertices are removed. Here, both vertices at the maxima v_{x^+} and v_{y^+} are removed. The new maxima must be on the cut boundary and are rounded to the appropriate integer coordinate.*

3.5.4. Axis-Aligned Bounding Box

In many cases the intermediate kernel $\hat{\mathcal{K}}$ is significantly smaller than the input mesh. Since many supporting planes do not intersect with $\hat{\mathcal{K}}$, we can use an axis-aligned bounding box (AABB) as a proxy. This allows us to apply a much faster intersection check with the AABB first and only need to check for a proper intersection with the intermediate kernel if the AABB is intersected. We define the AABB in integer coordinates with center c_{aabb} and half-extents $\vec{e} = (e_x, e_y, e_z)^\intercal$ such that the eight corners of the AABB lie at $c_{aabb} + (\pm e_x, \pm e_y, \pm e_z)$. To determine, if an integer cutting plane (a,b,c,d) intersects, only a single check adapted from [Sza17] is required:

$$\langle c_{aabb}, (a,b,c)^\intercal \rangle + \langle \vec{e}, |\vec{n}| \rangle + d \geq 0$$

In this case $|\vec{n}|$ is the component-wise absolute value of the normal vector of the plane. Since we want to check against an integer AABB, the center might not be a valid integer coordinate. To accommodate for this, we derive c_{AABB} and $\vec{e}$ from the minimal and maximal coordinates of the AABB. To avoid the division by two, we multiply all values in the check by two, requiring just a single additional bit. The entire check requires at most 88 bits, which is still far below the 256 bit limit of the exact arithmetic framework.

Initially, the AABB exactly contains the input mesh. Each time the intermediate mesh kernel is cut, we might want to update the AABB in case it has become smaller. The update process is shown in Figure 4: To avoid recomputing the AABB from scratch and iterating over the entire polyhedron, we track the minimum and maximum vertex $v_{D^-}, v_{D^+}, D \in \{x,y,z\}$ in each axis of $\hat{\mathcal{K}}$. Each time such an extreme vertex is removed from the polyhedron, we update the corresponding axis. By construction, we know that the new minimum/maximum must be on the cut boundary, and therefore we may only consider these vertices for the update. The new extreme coordinate is then rounded up or down in the appropriate axis direction to assure that the AABB is integer and contains $\hat{\mathcal{K}}$ entirely.

3.6. LP Kernel Existence Check

In practice, many real-world meshes have an empty mesh kernel. As we will see in the evaluation (Section 4), of the approximately 2700 meshes from the test dataset also used by [AS24a], only about 400 have a non-empty mesh kernel.

Therefore, it is beneficial to quickly determine if a mesh even has a non-empty mesh kernel before running the full algorithm. Asiler et al. [AS24a] adopt the approach of Berg et al. [DVO*97] which initially solves a linear program (LP) to determine if the input mesh has a non-empty mesh kernel. The inputs are the half-spaces of the supporting planes of the input mesh. If the linear program is infeasible, the kernel must be empty.

They apply Seidel's solver [Sei91] which on average scales linearly with the number of input planes for small dimensions if the inputs are randomized. This makes it well suited for 3D problems such as this one. Unfortunately, the commonly used implementation of the Seidel solver is not exact, and the output may be incorrect, leading to false positives or negatives regarding the feasibility of the problem. In Figure 11 we will showcase where the inexact solver used in [AS24a] fails to determine the correct existence of a mesh kernel. To address this, we propose to use the same exact plane-based integer arithmetic framework presented in [NTK21; TNK22] that we also use for the mesh kernel cutting algorithm (Section 3.4) to solve the LP exactly.

3.6.1. Seidel's Solver

To understand how we solve the LP in an exact fashion, we will first give a brief introduction on how Seidel's solver [Sei91] works:

Assuming, we want to solve a D-dimensional problem, then we have initial constraints in the form of n D-dimensional half-spaces $h_i, i \in 1 \ldots n$. A point v_s is a valid solution if it lies inside the intersection of all n half-spaces.

If we neglect degenerate cases for now, then the intersection of the first D planes results in a point, that is a valid solution $\hat{s}$ for the first D constraints $h_1 \ldots h_D$. We can now iterate through the remaining constraints $h_{D+1} \ldots h_n$: As long as the current solution $\hat{s}$ is valid for the next constraint h_i, we can continue. However, as soon as $\hat{s}$ is not valid anymore for h_i, we must find a new solution that is valid for both h_i and all previous constraints $h_j, j < i$. This is done by solving a $D-1$ dimensional problem where the constraints are the projections of the constraints that have already been satisfied $h_j, j \leq i$ into the plane of the invalidated constraint h_i. If the $D-1$ dimensional problem has a solution, we can lift it back into the D dimensional space and continue with the next constraint. If the $D-1$ dimensional problem is infeasible, the original problem is infeasible and the solver terminates. The $D-1$ dimensional problem is solved recursively in the same fashion as the original problem. Once the problem reaches $D = 1$, the problem becomes a simple interval check. If the interval is empty, the problem is infeasible and the solver terminates immediately; otherwise a point of the interval is a valid solution that is then again lifted back up and the process continues. Usually, LP solvers have a linear objective function that is optimized; in the $D = 1$ case, the interval boundary that maximizes the objective is lifted back up.

3.6.2. Exact Plane-Based Seidel Solver

In our case, we have a 3D problem, and we can use the same exact integer arithmetic framework as before (c.f. Section 3.4). Using the same arithmetic also guarantees that the results of the LP solver and the mesh cutting are always consistent. The core idea is that instead

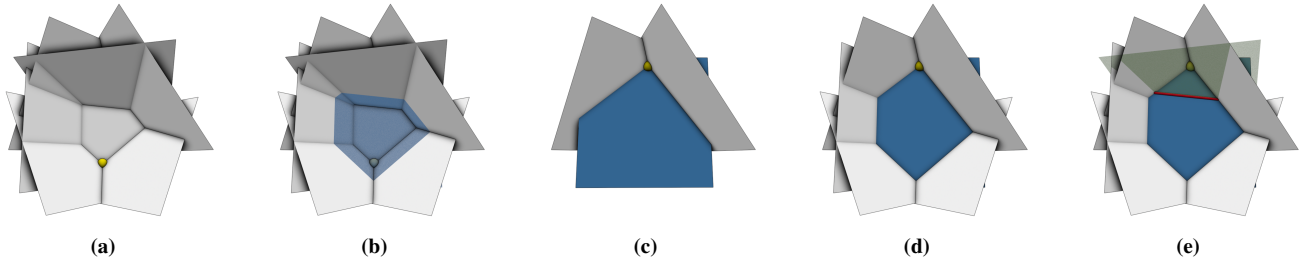


Figure 5: *(a) shows an intermediate state of our exact LP solver that has a valid 3D solution (yellow point, a) which the next constraint (blue, b) will invalidate. The invalidating constraint then becomes the fixed plane for the 2D problem. After an initial solution for the 2D problem is found by intersection with two previously processed planes from the 3D problem (c), further constraints are processed (d), until again, an invalidating constraint (green) is found (e). Then, a 1D problem is solved on the intersection line (red) of the invalidating constraints of the 3D and 2D problems.*

of projecting the problem into a lower-dimensional space, we can instead constrain the problem to a lower dimensional primitive that is still embedded into 3D space. An example of this is shown in Figure 5.

The input constraints $h_i, i \in 1 \dots n$ are the half-spaces of the supporting planes p_i of the input mesh, the same ones used for the mesh kernel cutting computation (Section 3.3). The current solution $\hat{s}$ is usually a point v_s given by the intersection of three planes $v_s = \texttt{intersect}(p_q, p_r, p_s), p, q, r \in 1 \dots n$. We start by finding an initial solution v_s in 3D. Once we find a half-space $h_{i_{3D}}$ that invalidates the current solution ($\texttt{classify}(v_s, h_{i_{3D}}) > 0$), we solve a 2D problem. For the 2D problem, we keep the original 3D constraints $h_j, j < i_{3D}$ but restrict the solution to the plane of $h_{i_{3D}}$. The 2D solution is therefore a 3D point consisting of three 3D planes $v_s = (p_{q'}, p_{r'}, p_{i_{3D}}), q', r' \in 1 \dots i_{3D}$. If the current 2D solution is invalid for a constraint $h_{i_{2D}}$, we solve a 1D problem. Now, we have the two planes p_{i2D} and p_{i3D} that define a line in 3D which is the domain of the 1D problem. A 1D solution is then a 3D point $v_s = (p_k, p_{i_{2D}}, p_{i_{3D}}), k \in 1 \dots i_{2D}$. Since all intermediate solutions are points in 3D, no lifting of the solution into a higher dimensional space is required.

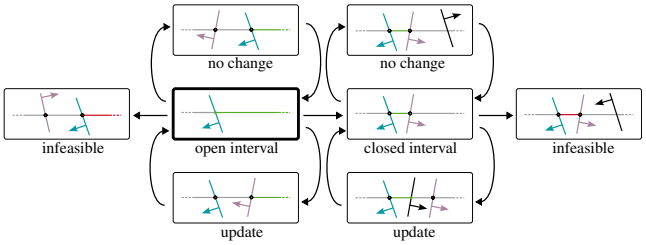


Figure 6: *For the 1D interval LP, only* `intersect` *and the* `classify` *predicate are used. The first plane that intersects with the interval line defines the left boundary of an open interval. Subsequent planes either update the left boundary, close the interval, or make the problem infeasible. Once a closed interval is found, new planes either contain the entire interval, update one of the boundaries, or the problem becomes infeasible.*

3.6.3. Exact 1D Interval Solver

For the 1D problem, we have to determine if the interval given by the constraints is empty or not. The entire interval handling is shown in Figure 6. Initially, the solution is the line of the first two fixed planes $p_{i_{3D}}$ and $p_{i_{2D}}$ from the 2D and 3D problem respectively. The first plane that intersects with the line defines the left interval boundary p_l and its corresponding intersection point $v_l = \texttt{intersect}(p_l, p_{i_{3D}}, p_{i_{2D}})$ with the line for an open boundary. Subsequent intersection planes p_i have intersection points $v_i = \texttt{intersect}(p_i, p_{i_{3D}}, p_{i_{2D}})$ with the line. We then compute $c_l = \texttt{classify}(v_l, p_i)$ and $c_i = \texttt{classify}(v_i, p_i)$. If both $c_l > 0$ and $c_i > 0$, the problem is infeasible, and the solver terminates. If $c_l \leq 0$ and $c_i \leq 0$, the interval becomes closed with the right boundary h_i. Otherwise, either a tighter left boundary is found and updated accordingly, or the open interval is still valid and the next half-space can be checked.

Once the interval is closed, the left and right intersection vertices are classified against the new half-space. If both are on the negative side, the interval is still a valid solution, nothing changes. If both are on the positive side, the problem is infeasible. Otherwise, the interval becomes smaller, and the bounds are updated accordingly. Planes that are parallel to the interval line either classify the entire line as valid or the problem as infeasible.

3.6.4. Initial Solution

In many cases, the initial 3D solution is the point $v_s = \texttt{intersect}(p_1, p_2, p_3)$, where p_1, p_2, p_3 are the first three input planes. However, sometimes the first three planes do not intersect due to coplanarity. To handle this, we define the initial solution $\hat{s}$ to be the entire surface of the first plane p_1. If our current solution is a plane p_s, subsequent coplanar constraints classify any point $v_s \in p_s$ against the constraint to check for validity of the current solution. The first non-coplanar constraint p_i will properly intersect with p_s. The new solution is then the line $\hat{s} = l_s = (p_s, p_i)$. If the current solution is a line, the first plane that properly intersects with it will update the current solution to their intersection point. Otherwise, coplanar constraints will classify any point on the line against the constraint. The same procedure is then applied for finding an initial solution for the 2D and 1D problems, but starts with the planes that were fixed in the higher-dimensional problems.

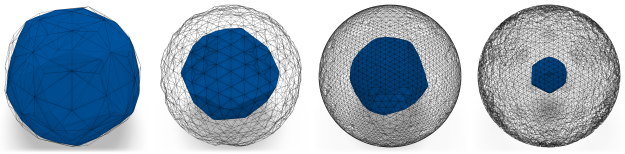


Figure 7: *Example models with their respective mesh-kernel from our artificial* star-shaped *dataset with increasing input complexity.*

3.6.5. Parallel LP Evaluation

Asiler et al. [AS24a] actually require the solution of the LP solver to find an initial point of the mesh kernel from which the remaining mesh kernel can incrementally be computed. Our approach does not require this, as we can directly compute the mesh kernel from the input mesh without the need for an initial point. We can still make use of the fact, that the LP solver will in many cases determine an empty kernel faster than our cutting algorithm by starting the LP solver in a parallel thread. If either the cutting or the LP solver determines that the mesh kernel is empty, we can terminate the computation immediately.

4. Evaluation

We have implemented our method in C++, compiled with GCC 12.2.0 and evaluated on a variety of 3D models. Its source code is available at https://github.com/juliusnehring/mesh-kernel. Unless stated otherwise, the experiments were conducted on a machine with an AMD Ryzen 9 5900X 12-core processor and 64 GB of RAM.

We have used two major datasets: The first dataset consists of a combination of real-world models. For this, we chose the same set of 2762 models used in [AS24a] consisting of all models from the Princeton Mesh Segmentation Benchmark [CGF09], as well as all closed, oriented genus 0 meshes from the Thingi10K dataset [ZJ16].

Since only about 400 of the 2700 models of the dataset contain a non-empty kernel, we have also generated an artificial dataset of star-shaped models that are guaranteed to have a non-empty kernel. To generate such models, we first generate an icosphere with varying number of subdivisions. Then, we randomly select a subset of the vertices and move them a random but bounded distance towards the center of the sphere. Some examples are shown in Figure 7. This dataset contains 10000 models. We will refer to the two datasets as *real-world* and *star-shaped* dataset, respectively.

4.1. Performance Comparison

We compare our method to the state-of-the-art method by Asiler et al. [AS24a], the method by Sorgente et al. [SBS22], as well as the implementation in CGAL [The24] that was also used for comparison in [AS24a]. For CGAL, we evaluate both an inexact arithmetic kernel `Simple_cartesian<double>` and an exact one `Exact_predicates_exact_constructions_kernel`. To guarantee a fair comparison, we disable our parallel evaluation of the exact LP solver, marked as *no LP*, because none of the other methods parallelize. We also evaluate the effect of the AABB proxy, marked as *no AABB* where all AABB optimizations are disabled, as well as disabling the cutting of planes adjacent to concave edges first, marked as *no con-first*. Similarly, we compare the effect collecting coplanar planes using a hash table (marked as *hash*) against using a union find data structure otherwise (cf. Section 3.5.2).

Method	Avg. (ms)	Med. (ms)	Total (s)
Polyhedron	37166	382.08	371668
CGAL (exact)	1848	169.28	18481
Kergen	357.7	18.71	3577.7
CGAL (inexact)	130.6	127.92	1150.3
Ours (hash)	40.49	15.32	404.9
Ours (no AABB)	38.43	12.28	384.4
Ours (no con-first)	35.08	14.13	350.8
Ours (no LP)	**32.20**	**5.085**	**322.0**

Table 1: *Timings for the artificial star-shaped dataset. Our method is on average an order of magnitude faster than the inexact Kergen [AS24a] and CGAL (inexact) [The24] and two orders faster than CGAL with exact arithmetic [The24] and even three orders faster than Polyhedron [SBS22].*

In Figure 9, we show the timings for computing the mesh kernel for the artificial dataset, both for input and output complexity. The corresponding statistics are shown in Table 1. We also show the timings for the real-world dataset in Figure 8 and Table 2. Our approach is about an order of magnitude faster than the inexact [AS24a]. In the upper plot of Figure 8, Polyhedron [SBS22] is faster for many models of the entire dataset. This is not reflected in the bottom plot, which only contains models that have a non-empty mesh kernel. This means that [SBS22] is often faster to determine if an input's kernel is empty, but slower in the actual kernel computation. Note, that because Polyhedron [SBS22] is not implemented in an exact manner, their method wrongly classifies some inputs with a non-empty kernel to having an empty one (cf. Table 3).

4.2. Exactness and Robustness

Since our method is exact, we are guaranteed to compute a valid kernel, if there is one. This is not always the case for inexact methods. For example, the inexact Seidel solver that is used in [AS24a] will sometimes determine the problem to be infeasible, even though there is a valid solution. For example, [AS24a] report that the models shown in Figure 11 do not have a kernel, while our exact method finds a valid one. Additionally, their method crashes for 12 models of the dataset, 10 of which have a valid kernel. Overall, for the real-world dataset, CGAL crashes for 314 models with an exact kernel, 359 times with an inexact kernel. Furthermore, our method is able to compute the kernel, even if it consists only of a single surface, line, or even just a single vertex as shown in Figure 10. Additionally, the exact representation used in our method also guarantees that it can be implemented robustly, meaning it never crashes for any given input.

To evaluate the quality of our implementation, we have compared our outputs to that of the other methods. The results are

Method	Avg. (ms)	Med. (ms)	Total (s)
CGAL (exact)	1829.880	80.718	4479.5
Polyhedron	1490.503	1.152	4103.4
CGAL (inexact)	1376.967	73.126	3318.5
Kergen	196.690	1.541	541.9
Ours (no AABB)	37.296	**0.613**	103.0
Ours (hash)	19.031	1.063	52.6
Ours (no con-first)	18.253	0.941	50.4
Ours (no LP)	17.722	0.868	48.9
Ours	**15.544**	0.630	**42.9**

Table 2: *Timings for the real-world dataset. Our method is about a magnitude faster than the inexact Kergen [AS24a] and about two magnitudes faster than CGAL with exact arithmetic [The24] and Polyhedron [SBS22]. We also compare the different optimizations of our method. Note that we compute the statistics over completed runs only, and for each method individually, as Kergen crashes 7 times, Polyhedron 9 times, CGAL (exact) 314 times, and CGAL (inexact) 352 times.*

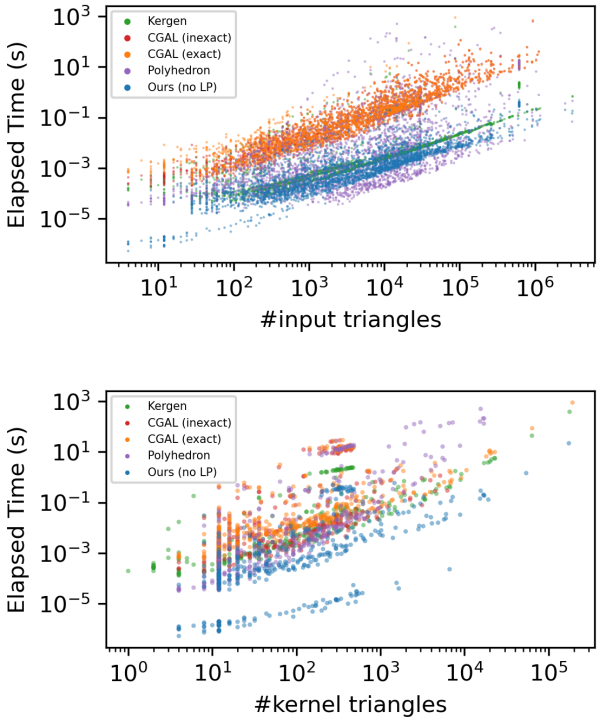


Figure 8: *Log-scale scatter plots of the timings for the real-world dataset. The top shows the time to compute the kernel or determine that there is none for the number of input triangles. The bottom shows the time to compute the kernel with the number of output faces on the x-axis.*

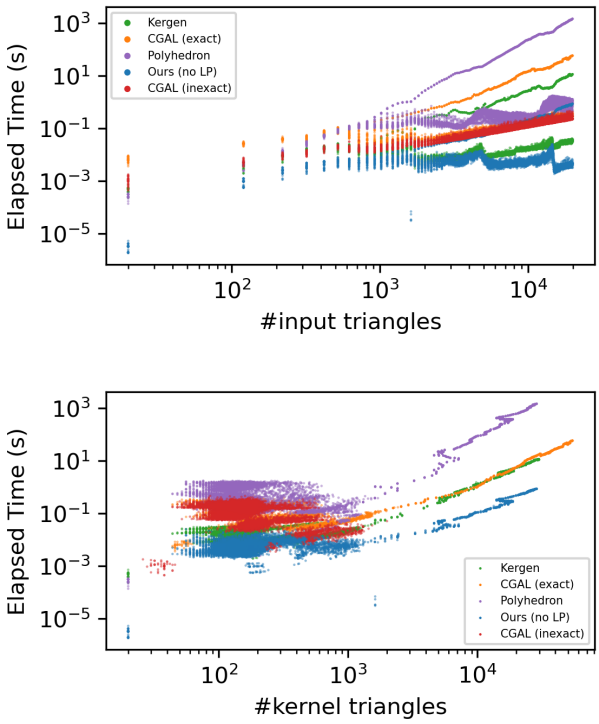


Figure 9: *Log-scale scatter plot of the timings for the artificial star-shaped dataset. The top shows the time to compute the kernel or determine that there is none for the number of input triangles. The bottom shows the time to compute the kernel with the number of output faces on the x-axis.*

Method	#kernels	$\epsilon > 0.1\%$	$\epsilon > 10\%$
CGAL (inexact)	341	118	21
Polyhedron	344	94	1
Kergen	362	2	40
CGAL (exact)	386	1	0
Ours	394	-	-

Table 3: *We have evaluated the number of kernels computed for the real-world dataset and compare the relative symmetric Hausdorff distance between our outputs and that of the other methods. Except for a single model and a few degenerate kernels, our outputs and those from CGAL (exact) [The24] are identical. Kergen [AS24a] computes a kernel for more models than Polyhedron [SBS22], but the kernel geometry is often significantly different from the exact solution. There are no cases, where a method computed a kernel, and ours did not.*

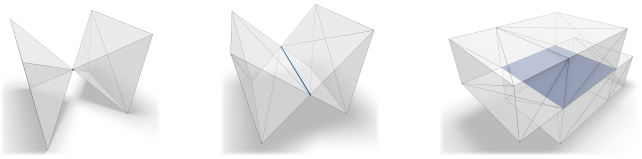


Figure 10: *The shown models are topologically genus 0 and have kernels that consist of a single vertex, edge, and face, respectively. Our exact method is able to compute these kernels, while inexact methods may fail.*

Figure 11: *Example models (69057, 237183, 423825) from the real-world dataset that have a non-empty but degenerate kernel that our method is able to compute. None of the previous methods [AS24a; The24; SBS22] are able to compute the kernel for these models.*

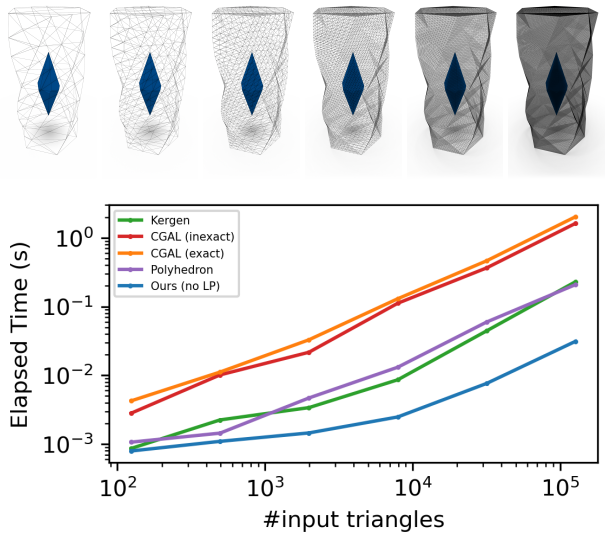


Figure 12: *We have reproduced the experiment from [AS24a]. The* spiral *mesh is subdivided only topologically. Note the logarithmic scale.*

shown in Table 3. To quantify the difference between the outputs, we have evaluated the relative symmetric Hausdorff distance between the output meshes, meaning we scale the combined AABB of both kernel meshes to the unit cube and compute the symmetric Hausdorff distance. We expect that our outputs are identical to those from CGAL (exact) [The24] since our method with 26 bit fixed point coordinates represents most floating point inputs (with 23 bits mantissa) exactly, and otherwise only produces marginal absolute error, and is fully exact after the initial conversion step. Except for a few inputs with degenerate kernels that CGAL (exact) [The24] fails to compute (cf. Figure 11 for some examples), and a single mesh where the outputs deviate less than 0.5%, both methods produce identical outputs. This single discrepancy (model 243 661) is topologically a quad mesh. Since the four vertices of a quad rarely lie exactly on the same plane, neither the representative plane, nor the triangulation of the quad is unique, resulting in two slightly different solutions. The error vanishes if the model is triangulated prior to being passed to both methods, as this common triangulation results in a unique set of planes.

This shows that our method computes the correct results when comparing against another exact method. Kergen [AS24a], Polyhedron [SBS22], and CGAL (inexact) [The24] produce significantly different outputs for many models.

4.3. Runtime Scaling

We have also reproduced the experiments from [AS24a] regarding the subdivided spiral and vase meshes. The spiral mesh (Figure 12) is subdivided topologically, which results in an almost constant sized kernel. Differences in the kernel size are due to numerical inaccuracies. The vase mesh (Figure 13) is subdivided geometrically, which results in a kernel that grows almost linearly with the number of input faces. The results are shown in Figure 12. Our method is significantly faster than the inexact method by Asiler et al. [AS24a] and the exact CGAL implementation [The24].

For Kergen [AS24b], it is possible that there are classes of inputs where the method eventually outperforms ours, as it scales only with $O(f \cdot r)$, where f is the number of input faces, and r is the number of output edges, whereas our method likely scales with $O(f \cdot log(f))$ according to Shamos and Hoey [SH76] as we compute the intersection of f halfspaces. Note that a non-empirical complexity comparison is difficult as Asiler et al. [AS24a] don't formally prove the average relationship between f and r, as well as that bounds lower than $O(f \cdot log(f))$ might be possible for the intersection of f halfspaces if their order is not random [SH76], which is the case for our method as we cut planes adjacent to concave edges first. Providing proofs for each method's complexity class is extremely challenging.

4.4. Plane Order

To evaluate the sensitivity of the runtime to the order in which the input planes are cut from the kernel, we have run our algorithm, but added the planes that will be part of the kernel first. This serves as an ideal oracle that gives us the planes first, that are guaranteed to be part of the output. We evaluated this on the real-world dataset, excluding convex models and those without a valid mesh kernel. The total time for the computation goes down by 20% from 35 to 28 seconds. Therefore, the order in which the planes are cut from the kernel provides only limited potential for further performance optimizations.

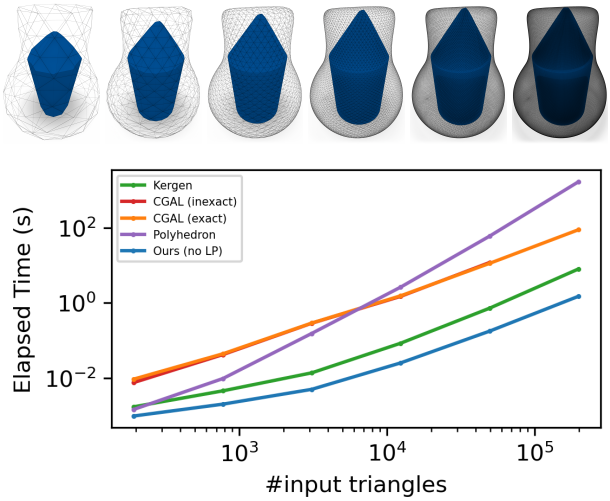


Figure 13: *We have reproduced the experiment from [AS24a] on the* vase *mesh with geometric subdivision. Due to the round shape of the vase, the kernel grows almost linearly with the number of input faces. CGAL (inexact) [The24] crashes on the final model. Note the logarithmic scale.*

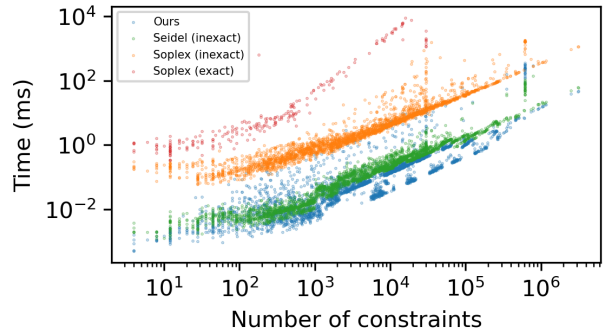


Figure 14: *Comparison for different LP solver implementations for our low-dimensional real-world dataset. Our exact LP Seidel solver is usually comparable in speed, if not faster, than the non-exact Seidel solver [Sei91]. It is about an order of magnitude faster than the non-exact simplex based Soplex [Wun96]. The exact Soplex solver [GSW12; GSW16] is significantly slower and often runs into a time limit of 10 seconds. Note the logarithmic scale.*

4.5. LP Solver

We have also evaluated the performance of our exact LP solver in comparison to the non-exact Seidel solver [Sei91] and the non-exact simplex based Soplex [Wun96], as well as an exact version of Soplex using rational numbers [GSW12; GSW16]. The results are shown in Figure 14. Our exact LP solver is at least competitive compared to the non-exact Seidel solver and significantly faster than both the non-exact and exact Soplex solvers. For soplex (exact), we introduced a time limit of 10 seconds after which the solver is terminated. The slowest solve of all other methods took 1.8 seconds. Note, however, that Seidel's solver is by design very efficient for low-dimensional problems, while Soplex is based on simplex, which generally scales better for higher dimensional problems.

5. Conclusion

We have presented a method to compute the kernel of an input triangle mesh. In contrast to previous methods, our approach is exact and robust while maintaining a high performance. We solve the problem by making use of exact plane-based integer arithmetic to both exactly compute the mesh kernel as the intersection of the input half-spaces, as well as efficiently determine whether a kernel exists by proposing an exact plane-based LP solver. Our evaluation has shown that our method is robust, which is not the case for previous methods, and is on average significantly more efficient.

For future directions there is some limited potential for improvement by the choice of which planes should be cut next. More importantly, none of the existing methods, including ours, parallelize well. However, we believe there are at least some opportunities for parallelization: Since many of the planes do not intersect with the kernel, one could check multiple planes in parallel and eliminate those that do not intersect with the kernel. This process can be repeated until all planes have either been eliminated or intersected with the kernel.

Acknowledgements

This work was supported by the DFG project FOR 5492 Polytope Mesh Generation and Finite Element Analysis Methods for Problems in Solid Mechanics.

References

[ACDE07] APPLEGATE, DAVID L, COOK, WILLIAM, DASH, SANJEEB, and ESPINOZA, DANIEL G. "Exact solutions to linear programming problems". *Operations Research Letters* 35.6 (2007), 693–699. DOI: 10.1016/j.orl.2006.12.010 2.

[AS24a] ASILER, MERVE and SAHILLIOĞLU, YUSUF. "KerGen: a kernel computation algorithm for 3D polygon meshes". *Computer Graphics Forum*. Vol. 43. 5. Wiley Online Library. 2024, e15137. DOI: 10.1111/cgf.15137 1, 2, 5, 7–10.

[AS24b] ASILER, MERVE and SAHILLIOĞLU, YUSUF. "3D geometric kernel computation in polygon mesh structures". *Computers & Graphics* 122 (2024), 103951. DOI: 10.1016/j.cag.2024.103951 2, 9.

[Att20] ATTENE, MARCO. "Indirect predicates for geometric constructions". *Computer-Aided Design* 126 (2020), 102856. DOI: 10.1016/j.cad.2020.102856 2.

[BF09] BERNSTEIN, GILBERT and FUSSELL, DON. "Fast, exact, linear booleans". *Computer Graphics Forum*. Vol. 28. 5. Wiley Online Library. 2009, 1269–1278. DOI: 10.1111/j.1467-8659.2009.01504.x 2.

[CGF09] CHEN, XIAOBAI, GOLOVINSKIY, ALEKSEY, and FUNKHOUSER, THOMAS. "A Benchmark for 3D Mesh Segmentation". *ACM Transactions on Graphics (Proc. SIGGRAPH)* 28.3 (Aug. 2009). DOI: 10.1145/1531326.1531379 7.

[CK10] CAMPEN, MARCEL and KOBBELT, LEIF. "Exact and robust (self-) intersections for polygonal meshes". *Computer Graphics Forum*. Vol. 29. 2. Wiley Online Library. 2010, 397–406. DOI: 10.1111/j.1467-8659.2009.01609.x 2.

[DFK*03] DHIFLAOUI, MARCEL, FUNKE, STEFAN, KWAPPIK, CARSTEN, et al. "Certifying and repairing solutions to large lps how good are lp-solvers?": *SODA*. Vol. 3. 2003, 255–256. URL: http://graphics.stanford.edu/~sfunke/Papers/SODA03B/LP_Exact_Camera.pdf 2.

[DVO*97] DE BERG, MARK, VAN KREVELD, MARC, OVERMARS, MARK, et al. "Computational geometry: introduction". *Computational geometry: algorithms and applications* (1997), 1–17. DOI: 10.1007/978-3-662-03427-9_1 5.

[GSW12] GLEIXNER, AMBROS M, STEFFY, DANIEL E, and WOLTER, KATI. "Improving the accuracy of linear programming solvers with iterative refinement". *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*. 2012, 187–194. DOI: 10.1145/2442829.2442858 10.

[GSW16] GLEIXNER, AMBROS M, STEFFY, DANIEL E, and WOLTER, KATI. "Iterative refinement for linear programming". *INFORMS Journal on Computing* 28.3 (2016), 449–464. DOI: 10.1287/ijoc.2016.0692 10.

[HBC24] HINDERINK, STEFFEN, BRÜCKLER, HENDRIK, and CAMPEN, MARCEL. "Bijective volumetric mapping via star decomposition". *ACM Transactions on Graphics (TOG)* 43.6 (2024), 1–11. DOI: 10.1145/3687950 2.

[HC23] HINDERINK, STEFFEN and CAMPEN, MARCEL. "Galaxy maps: Localized foliations for bijective volumetric mapping". *ACM Transactions on Graphics (TOG)* 42.4 (2023), 1–16. DOI: 10.1145/3592410 2.

[Koc03] KOCH, THORSTEN. "The final NETLIB-LP results". (2003). URL: https://opus4.kobv.de/opus4-zib/files/727/ZR-03-05.pdf 2.

[Liv24] LIVESU, M. "Advancing Front Surface Mapping". *Computer Graphics Forum* 43.2 (2024), e15026. DOI: 10.1111/cgf.15026 2.

[LP79] LEE, DER-TSAI and PREPARATA, FRANCO P. "An optimal algorithm for finding the kernel of a polygon". *Journal of the ACM (JACM)* 26.3 (1979), 415–421. DOI: 10.1145/322139.322142 2.

[NTK21] NEHRING-WIRXEL, JULIUS, TRETTNER, PHILIP, and KOBBELT, LEIF. "Fast exact booleans for iterated CSG using octree-embedded BSPs". *Computer-Aided Design* 135 (2021), 103015. DOI: 10.1016/j.cad.2021.103015 2–5.

[PM79] PREPARATA, FRANCO P. and MULLER, DAVID E. "Finding the intersection of n half-spaces in time O (n log n)". *Theoretical Computer Science* 8.1 (1979), 45–55. DOI: 10.1016/0304-3975(79)90055-0 2.

[Ric97] RICHARD SHEWCHUK, JONATHAN. "Adaptive precision floating-point arithmetic and fast robust geometric predicates". *Discrete & Computational Geometry* 18 (1997), 305–363. DOI: 10.1007/PL00009321 2.

[SBMS22] SORGENTE, TOMMASO, BIASOTTI, SILVIA, MANZINI, GIANMARCO, and SPAGNUOLO, MICHELA. "The role of mesh quality and mesh quality indicators in the virtual element method". *Advances in Computational Mathematics* 48.1 (2022), 3. DOI: 10.1007/s10444-021-09913-3 2.

[SBS22] SORGENTE, TOMMASO, BIASOTTI, SILVIA, and SPAGNUOLO, MICHELA. "Polyhedron kernel computation using a geometric approach". *Computers & Graphics* 105 (2022), 94–104. DOI: 10.1016/j.cag.2022.05.001 1–3, 7–9.

[Sei91] SEIDEL, RAIMUND. "Small-dimensional linear programming and convex hulls made easy". *Discrete & Computational Geometry* 6 (1991), 423–434. DOI: 10.1007/BF02574699 5, 10.

[SH76] SHAMOS, MICHAEL IAN and HOEY, DAN. "Geometric intersection problems". *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE. 1976, 208–215. DOI: 10.1109/SFCS.1976.16 2, 9.

[Sza17] SZAUER, GABOR. *Game Physics Cookbook: Discover over 100 easy-to-follow recipes to help you implement efficient game physics and collision detection in your games*. Packt Publishing Ltd, 2017, 190–192 5.

[Tar75] TARJAN, ROBERT ENDRE. "Efficiency of a Good But Not Linear Set Union Algorithm". *J. ACM* 22.2 (Apr. 1975), 215–225. ISSN: 0004-5411. DOI: 10.1145/321879.321884 4.

[The24] THE CGAL PROJECT. *CGAL User and Reference Manual*. 6.0.1. CGAL Editorial Board, 2024. URL: https://doc.cgal.org/6.0.1/Manual/packages.html 2, 7–10.

[TNK22] TRETTNER, PHILIP, NEHRING-WIRXEL, JULIUS, and KOBBELT, LEIF. "EMBER: exact mesh booleans via efficient & robust local arrangements". *ACM Transactions on Graphics (TOG)* 41.4 (2022), 1–15. DOI: 10.1145/3528223.3530181 2, 3, 5.

[Wol03] WOLF, JOHN P. *The scaled boundary finite element method*. John Wiley & Sons, 2003 1.

[Wun96] WUNDERLING, ROLAND. "Paralleler und objektorientierter simplex-algorithmus". MA thesis. 1996 10.

[XZWC24] XU, KAI, ZHENG, LANXIANG, WEI, MINGXIN, and CHENG, HUI. "VRExplorer: An Efficient View-Region based Autonomous Exploration Method in Unknown Environments for UAV". *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2024, 8081–8087. DOI: 10.1109/IROS58592.2024.10802710 1, 2.

[YL11] YU, WUYI and LI, XIN. "Computing 3d shape guarding and star decomposition". *Computer Graphics Forum*. Vol. 30. 7. Wiley Online Library. 2011, 2087–2096. DOI: 10.1111/j.1467-8659.2011.02056.x 1, 2.

[ZJ16] ZHOU, QINGNAN and JACOBSON, ALEC. "Thingi10K: A Dataset of 10,000 3D-Printing Models". *arXiv preprint arXiv:1605.04797* (2016). DOI: 10.48550/arXiv.1605.04797 7.