# A Simple and Fast Hole Detection Algorithm for Triangulated Surfaces

Kumar, S. Vijai; Vuik, Cornelis

**DOI**
10.1115/1.4049030

**Publication date**
2021

**Document Version**
Final published version

**Published in**
Journal of Computing and Information Science in Engineering

# A Simple and Fast Hole Detection Algorithm for Triangulated Surfaces

**S. Vijai Kumar**
Delft Institute of Applied Mathematics,
Delft University of Technology,
Delft 2628 XE, Netherlands
e-mail: v.k.suriyababu@tudelft.nl

**Cornelis Vuik**
Professor
Delft Institute of Applied Mathematics,
Delft University of Technology,
Delft 2628 XE, Netherlands
e-mail: c.vuik@tudelft.nl

*We present a simple and fast algorithm for computing the exact holes in discrete two-dimensional manifolds embedded in a three-dimensional Euclidean space. We deal with the intentionally created "through holes" or "tunnel holes" in the geometry as opposed to missing triangles. The algorithm detects the holes in the geometry directly without any simplified geometry approximation. Discrete Gaussian curvature is used for approximating the local curvature flow in the geometry and for removing outliers from the collection of feature edges. We present an algorithm with varying degrees of flexibility. The algorithm is demonstrated separately for sheets and solid geometries. This article demonstrates the algorithm on triangulated surfaces. However, the algorithm and the underlying data structure are also applicable for surfaces with mixed polygons.* [DOI: 10.1115/1.4049030]

*Keywords: computational geometry, computer-aided design, computer-aided engineering*

## 1 Introduction

The problem of hole detection and hole filling has been around for a long time in geometry processing and repair. Many numerical simulations use simplified geometries and require manual defeaturing. In geometries with hundreds of thousands of holes, this can be a very daunting task [1]. There is a great deal of necessity to detect these holes in the geometry for applications such as automation of computational fluid dynamic simulations. Since the holes in some cases form the inlets and outlets of the geometry, it is also quite useful in applications such as volumetric mesh generation where it is undesirable to have leaky geometries [2]. Doraiswamy et al. [3] proposed an algorithm to compute Reeb graphs of a discrete surface by solving a scalar-valued function to compute the critical points on a surface. These algorithms are very robust in computing the genus of the surface and work for a broad range of geometries. However, they are also susceptible to minor disturbances in the surface and can lead to far too many volumes. It is a very costly process and requires a more substantial computational time compared to the algorithms proposed in this article. Guillem Borrell et al. [4] proposed an algorithm that relies on a voxelized mesh for computing the genus of the surface. Hence, it is impossible to extract the original geometry of the hole. The literature discussed earlier, albeit its marginal similarity, do not aim to find the holes in the geometry.

Smereka et al. used a modified Hough transform for detection of circular objects in point clouds [5]. They accomplish it by looking for inliers and outliers of a circle using the parametric equation of a circle. This algorithm can give an approximation of point clusters that could be circles and cylinders. The only articles, as far as we know, which attempts to solve the problem of detecting holes in discrete surfaces are algorithms proposed by Wang et al. [6] and Lozano-Durán and Borrell [4]. They rely on the coplanarity of the triangles to merge triangles to form multiple planes. They group adjacent planes into cluster groups of volumes for extracting contours of holes. It works very well for simple and relatively complex surfaces, which have some inherent planarity. For industrial cases, they end up forming too many planes, and the authors themselves suggest some surface segmentation-based approaches. The present investigation highlights the application of algorithms for multiple practical scenarios for sheet and solid geometries. The algorithm only relies on the feature angle and discrete Gaussian curvature for extracting the holes from the surface. We designed our algorithm in such a way that it does not require any removal of topological elements from the data structure. Thus, it is possible to fully parallelize all the steps of the algorithm except the loop formation algorithm. It also does not require any voxelization or any statistical approximation and is capable of extracting the accurate geometry of the detected holes. Nevertheless, our algorithm is not a replacement to above approaches. It complements these algorithms. Our approach can detect most of the holes in complex geometries, but they can also overpredict or underpredict them as shown later due to their semi-heuristic nature. We aim to complement approaches such as Reeb graphs since the Reeb graph-based approaches also provide information about internal volumes.

## 2 Essential Definitions and Nomenclature

We start with the basic topological elements of a discrete surface. The algorithms proposed are valid for any polygonal surface. However, the discussion here is restricted to triangulated surfaces due to their commonality. A triangulated surface hierarchy is as follows.

```
surface
        Triangle 1
                Vertex 1
                Vertex 2
                Vertex 3
        Triangle 2
                Vertex 1
                Vertex 2
                Vertex 3
        and so on…
```

The order of the edges determines the orientation of the surface, and it is imperative to maintain consistency throughout the domain. A well-known mesh data structure (half-edge [7]) forms the base of all algorithms. The half-edge data structure from a robust open-source library called OpenMesh [8] is used for all algorithms. The choice of OpenMesh over more popular alternatives like CGAL [9] is due to CGAL's restriction on any nonmanifold elements (vertices, edges, or faces). OpenMesh allows the algorithmist to deal with these nonmanifold elements how they see it fit. OpenMesh is also released under a more friendly open source license and has zero dependencies.

**2.1 Half-Edge Data Structure.** In simple terms, half-edge data structure splits every edge in a surface into two half-edges with opposite orientation allowing every face of a polygon to have its unique edges (as opposed to sharing an edge between two faces). Traversal across the surface is very easy with this data structure. The pictorial representation in Fig. 1 shows the different components of a half-edge data structure. Each half-edge stores its originating (or incoming vertex) and points to an outgoing vertex. Since a half-edge is an edge split into two, each half-edge also stores its pair allowing one to find the adjacent faces of a triangle

Manuscript received July 31, 2020; final manuscript received October 27, 2020; published online February 11, 2021. Assoc. Editor: Charlie C. L. Wang.

with just a half-edge. Each vertex stores its incoming half-edge and outgoing half-edge. However, this is not very strict, and the data structure is adapted by algorithmists to store more/less information based on their preference. There are also several iterators and circulators written for iterating over all incident edges of a vertex, face, and edge. Most implementations of half-edge data structures also allow storing custom properties on mesh entities.

**2.2 Discrete Gaussian Curvature.** Curvature flow of a surface, in general, can be computed only on continuous surfaces. However, most of the practical applications in computer-aided design and computer graphics deal with discrete surfaces. Hence, it is important to define the curvature flow of a surface in a discrete form [10]. The Gaussian curvature of a discrete edge for a vertex in a surface can be computed by the following relation [10], where $V$ denotes a Vertex, $i$ denotes the incident vertices, $\theta$ denotes the incident angle, and $K$ denotes the discrete Gaussian curvature.

$$K_V = \sum_V [2\pi - \sum_i \theta_i] \tag{1}$$

The algorithm for computing the same on a half-edge data structure is shown below.

**Algorithm 1** Computation of discrete Gaussian curvature (one vertex)

**Data**: Vertex id (Unsigned Integer/Integer)
**Result**: Discrete Gaussian curvature of an input vertex (Double)
**if** *Input vertex is deleted/does not exist* **then**
    **return** 0.0;
**end**
Initialize input vertex as Reference vertex;
Initialize *GaussianCurvature* as $2\pi$;
Get a reference to an outgoing half-edge iterator for Input vertex as *voh_it*;
**if** *Outgoing half-edge iterator is invalid* **then**
    **return** 0.0;
**end**
Get a reference to next outgoing half-edge iterator for Input vertex as *n_voh_it*;
**forall** *Outgoing half-edges of an input vertex* **do**
    $k = k - cos^{-1} \frac{|| CurrentOutgoingHalfEdge - ReferenceVertex ||}{|| NextOutgoingHalfEdge - ReferenceVertex ||}$
**end**
**return** $k$;

The discrete Gaussian curvature allows to identify the curvature of a surface locally (for a vertex/edge/face).

| Gaussian curvature | Local surface vertex classification |
|---|---|
| $>0$ | Elliptic |
| $<0$ | Hyperbolic |
| $=0$ | Parabolic |

The discrete Gaussian curvature is used as an initial filtering metric to improve and accelerate the convergence of the algorithm. Only the vertices classified as possibly a *hyperboloid* or a *cylinder* are kept, and rest of the vertices are filtered out from the search space. The primary advantage of Gaussian curvature is that it gives information about the planar areas in a surface so that they can be removed from the search space. The choice of filtering

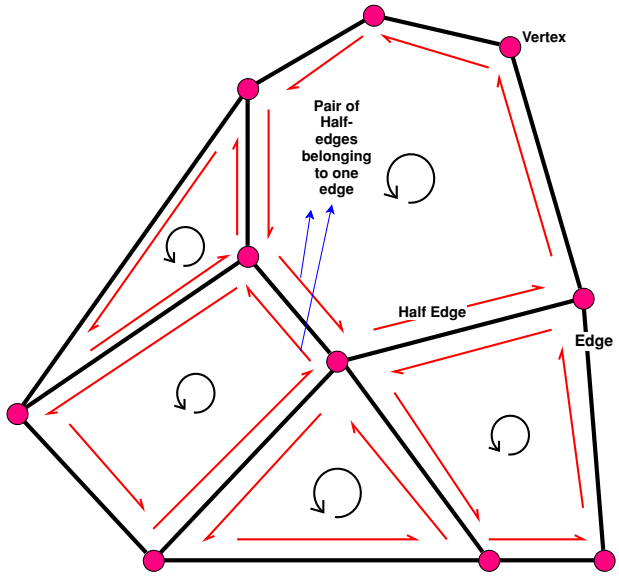


**Fig. 1 Half-edge data structure represented on a polygonal mesh**

was based on experimental observation and also based on the fact the Gaussian curvature helps identify the planar areas of the mesh. This metric is also quite useful for parameterization and remeshing algorithms.

**2.3 Feature Angle.** Feature angle is the primary and starting filtering criterion for all the algorithms discussed in this article (Fig. 2). The angle between any two polygons is the feature angle of the shared edge connecting them. The choice of the angle is the same (between 30 and 120 deg) throughout the article based on experimental observation. Ideally, the edges forming the hole should form an angle of 90 deg. Real-world meshes have disturbances. Hence, our choice is a bit more conservative to ensure a higher success rate. However, this angle range can be expanded depending on the mesh at hand. The increase in this range would increase the size of the search space and slow down the algorithm considerably. We have found out that a choice of 30–120 deg worked in all of our test cases.

**2.4 Hole.** The term "hole" is more commonly used in the literature for missing polygons (triangles in this case) in a discrete surface. There have been several algorithms proposed on the discovery and closing of holes in discrete surfaces [11,12]. However, the missing triangles in geometry are far too easy to detect. This article does not focus on the holes mentioned in the context of these pieces of the literature. We focus on intentionally created holes whose features are desired in a geometry. The algorithm discussed here is more along the lines of detecting the genus of a discrete surface. The algorithms proposed in this article can be modified to identify the approximate genus of a surface. It could also be reduced to a problem of detecting closed-loop features in a geometry. The geometries created in computer-aided design are quite complex and often contain many holes, fillets, and chamfers among other sophisticated features.

## 3 Algorithm

The algorithm for the detection of holes is discussed in detail here. These algorithms are created as a result of a top-down approach. We started with a brute force approach that requires searching the entire search space of the geometry for holes. Then, the goal is to reduce the search space with suitable

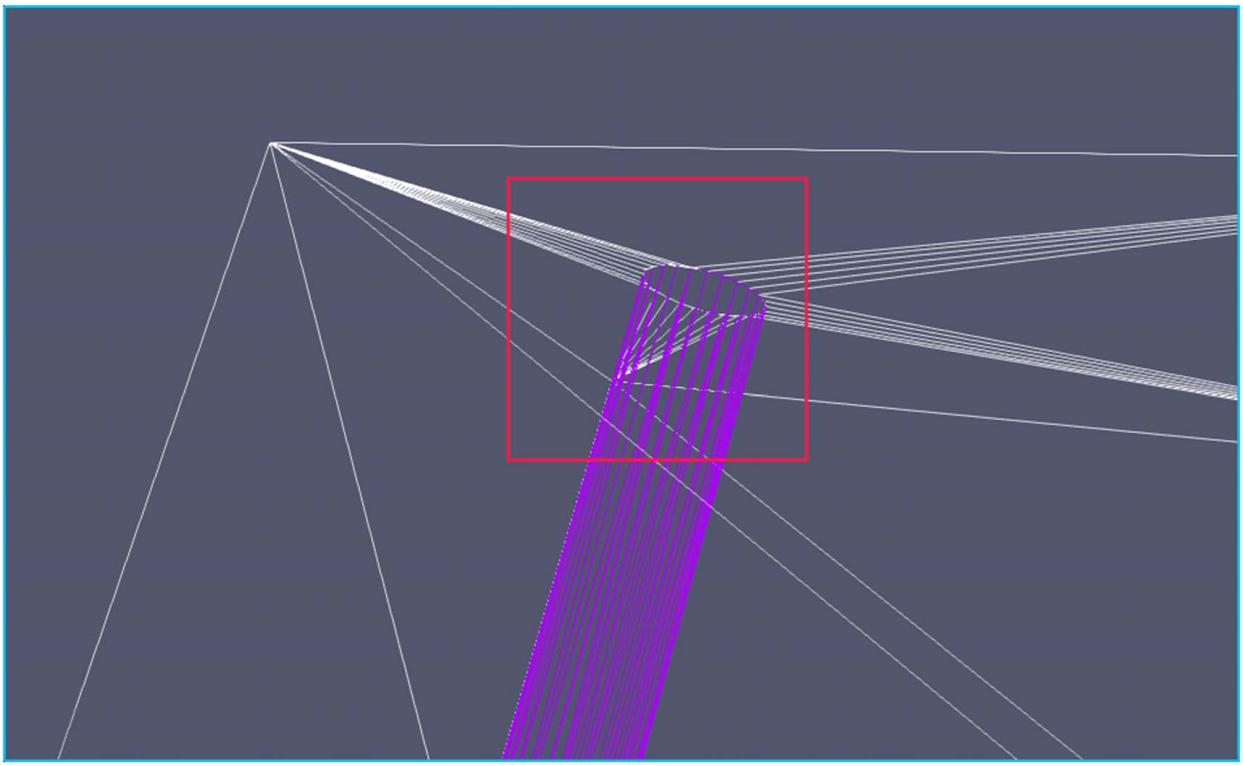

**Fig. 2 Feature angle (the angle between faces of cylinder and adjacent faces form the feature angle)**

geometric filters. We started adding additional filtering metrics such as feature angle and Gaussian curvature to reduce the search space. Hence, the algorithm is semi-heuristic in nature. Experimental results shown in the subsequent sections are done with real geometries used in finite element analysis (FEA)/computational fluid dynamics simulations. In this article, we discuss two different algorithms. The most important algorithm is the *fast algorithm for detection of holes accelerated with discrete Gaussian computation*. We also show a comparison of this algorithm with a purely brute force approach in Appendix A of this article.

(1) Fast algorithm for detection of holes accelerated with discrete Gaussian computation
(2) Detection of holes in sheets

For a given mesh represented as triangulation, the triangles are first loaded into the half-edge data structure. The mesh reader fixes some of the topological issues in the mesh (such as nonmanifold elements). We compute its feature angle of every triangle with its three neighbors. The triangles that satisfy our feature angle criteria are marked accordingly. As a next step, Gaussian curvature is computed on the mesh, and the planar areas are filtered out by marking them with property traits. We start searching for looped features in this reduced search space and then pair them up or remove them depending on the requirements.

**Algorithm 2** Algorithm to detect holes in solids (Gaussian-filtered search)

**Result**: Vector of Chains (Every chain is a closed loop of edges)
Initialize Surface into a half-edge data structure;
Create an Empty vector called **possible_edges** to store edges;
**forall** *Faces of Surface* **do**
  Get the three surrounding faces;
  Compute the feature angle between the current face and its neighbors;
  **if** *The feature angle matches a specific feature angle criterion* **then**
    add them to possible_edges vector;
  **end**
**end**
**forall** *Edges in possible_edges vector* **do**
  Compute the discrete Gaussian curvature;
**end**
**forall** *Edges in possible_edges vector* **do**
  **if** *Edge is **NOT** possibly part of a hyperboloid or cylinder* **then**
    Remove the edge from possible_edges vector;
  **end**
**end**
**while** *All edges in possible_edges vector is equal to 2* **do**
  Remove edges which do not have the matching valence;
**end**
Loop through **possible_edges vector** and form chains;

**3.1 Fast Algorithm for Detection of Holes Accelerated With Discrete Gaussian Computation.** The first algorithm is the fastest and most robust algorithm. It is very efficient at detecting most of the holes in a geometry. It relies on the feature angle of the triangles and the discrete Gaussian curvature for detecting the holes in the geometry. It fails in case of very poorly meshed geometries with bad feature angles. Examples of such cases are shown in Sec. 4 of this article. The algorithm listing is as follows.

**3.2 Hole Detection in Sheets.** Detection of holes in sheets is relatively trivial. It is very similar to the detection of missing triangles (also known as holes) in solids. The half-edge data structure takes care of most of the work in this case. Since a sheet geometry has a clearly defined border, this reduces to a simple border edge detection problem. The border edge or edges forming hole will not have two pairs of half-edges. Instead, it will have the same half-edge pointing to itself. The algorithm collects these half-edges and

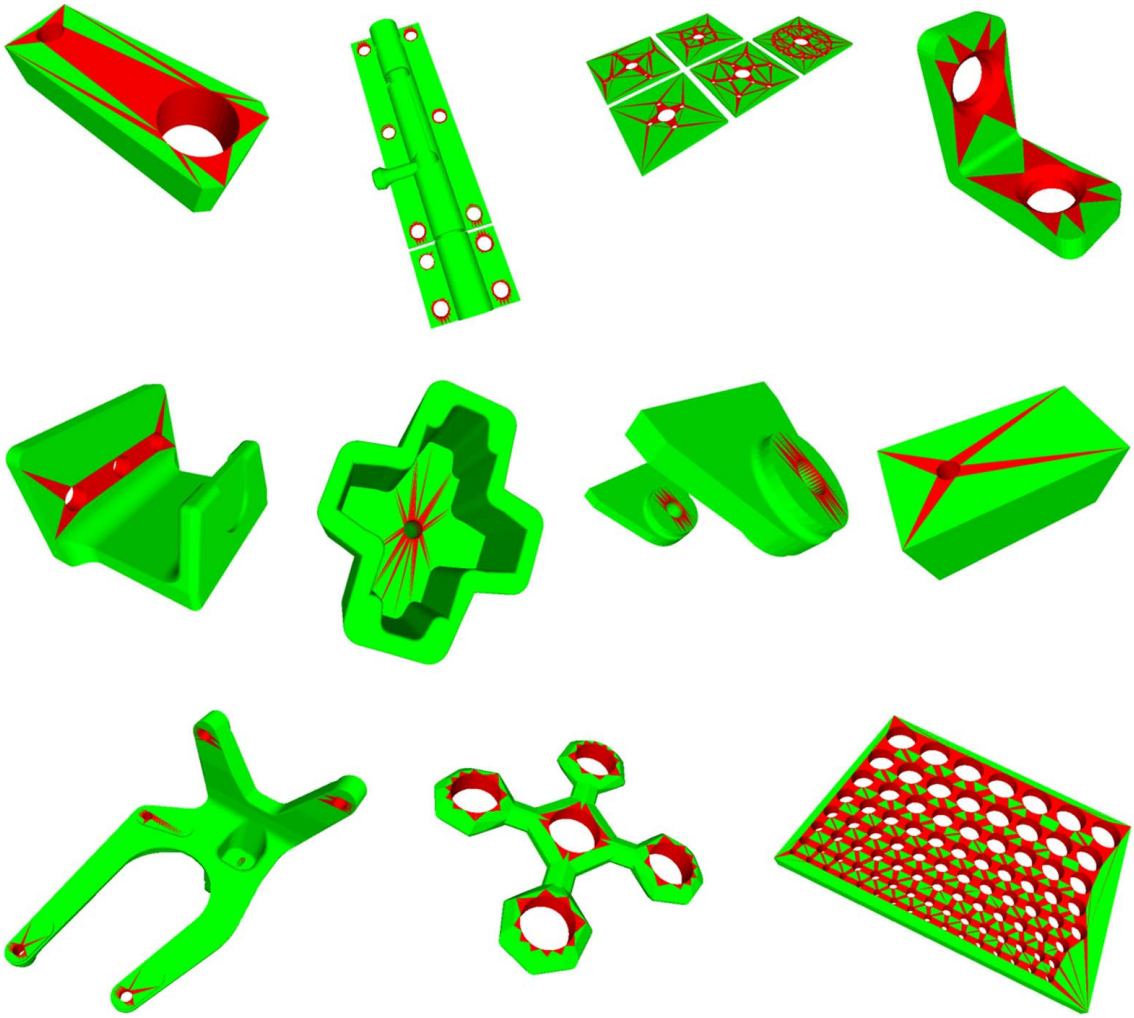

**Fig. 3 Holes detected by algorithm in different geometries (colored by hole faces—dark color indicates the faces forming the hole and light color indicates the faces that are not part of a hole)**

forms loops. It requires a simple pruning process to remove unnecessary holes, especially the ones formed by a few missing triangles. The same algorithm from solids can be used for the formation of loops in sheet geometries.

**Algorithm 3** Hole detection in sheet geometries

```
Data: Discrete surface
Result: Vector of Chains (Every chain is a closed loop of edges)
Initialize Surface into a half-edge data structure;
Create an Empty vector called boundary_edges to store edges;
forall Faces of Surface do
    forall Half Edges in a Face do
        if If pair of Half Edge is itself then
            Add the edge to the boundary_edges vector;
        end
    end
end
Loop through boundary_edges vector and form chains;
```

**3.3 Formation of Chains (or Closed Loops).** The algorithm starts with an edge from the reduced search space and chooses to collect edges either in a clockwise or counterclockwise direction. Since the algorithm relies on the half-edge data structure, complete connectivity is already established and does not require any complex algorithm to find the next edge. The chain formation algorithm collects edges and adds them into a vector until it circles back to the original edge. If there is a discontinuity somewhere in a loop, then the entire sequence of edges is wholly removed (since the loop is not closed anymore). The removal is only virtual (accomplished by property traits and bookkeeping data structures).

**3.4 Approximate Diameter and Removal.** We compute the approximate diameter of a hole with the help of the mini ball algorithm [13]. This algorithm helps to match pairs of holes of the same radius quickly. A hole can be removed in most cases quite merely by the removal of the layer connecting the top and bottom hole and by triangulating the hole using any simple triangulation algorithm.

## 4 Experiments

We tested the hole detection algorithm for sheets and solids on geometries with different degrees of complexity (Fig. 3).

C++11 and PYTHON were used for code implementation and visualization, respectively. A laptop with Intel Core i7-8700K CPU @ 3.70GHz CPU on a single core with 64 gigabytes of RAM is used for benchmarks. All results are shown for the Gaussian-filtered search algorithm, and the combined runtime for all geometries listed here is less than 5 s. The algorithm scales linearly in time

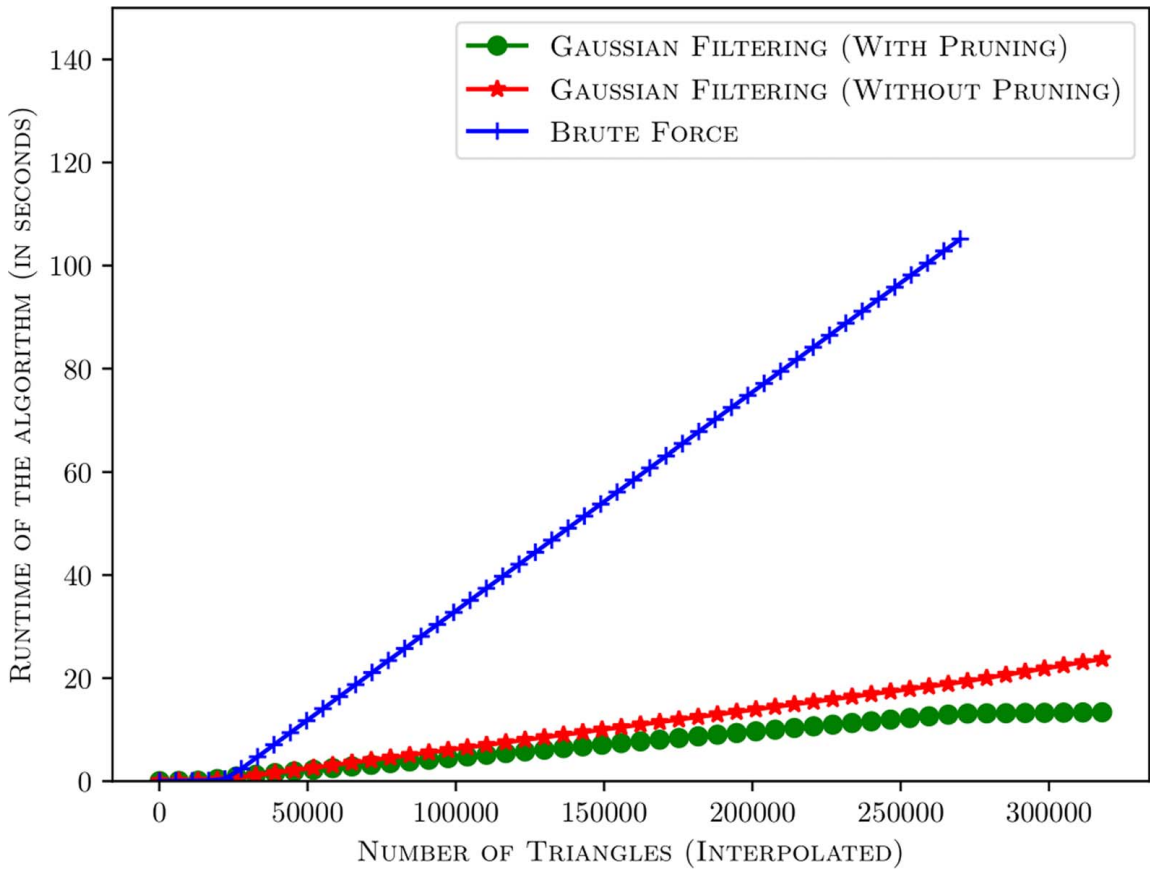


**Fig. 4 Runtime analysis (brute force algorithm does not work/gives erroneous results for larger test cases)**

with the number of triangles. We experimented for meshes with an increasing number of triangles. Different components of the algorithm such as edge pruning or Gaussian-filtered search could be enabled or disabled to form own variants of the algorithm depending on use cases. The Gaussian-filtered search algorithm (with and without pruning) is compared against its brute force counterpart, and their runtimes are shown in Fig. 4. It can be observed that for the Gaussian-filtered search (with and without pruning) algorithm, the runtime scales linearly with the number of triangles, whereas for the brute force algorithm, the growth is exponential. In complex cases, as shown in Fig. 4, the brute force approach either gives erroneous results or is unable to handle the input altogether. This analysis shows the feasibility of Gaussian-filtered search algorithms for practical purposes. The runtime can further be improved by parallelization of the majority of the code. Except for the loop formation algorithm, the rest of the algorithm is embarrassingly parallel and can obtain high speedup with libraries like OPENMP. We did not perform any strong/weak scaling analysis since its a purely serial algorithm. Our algorithm scales linearly in time with an increase in the number of triangles.

In case our algorithm fails to detect any holes, the algorithm for sheets is run on the geometry to check if it can find any holes before termination.

## 5 Conclusion

We proposed a set of algorithms to detect holes in discrete surfaces. We showed with experiments and experimental run time analysis that the usage of discrete Gaussian curvature as a filtering metric outperforms the brute force search algorithm by simplifying the search space. The local curvature information allows the algorithm to remove sharp edges in the search space that can be easily removed later during the pruning process. The exhaustive (or brute force) search algorithm is also useful in cases where geometry is simply of inferior quality, lacks good feature angles, or has many disturbances near the holes. The algorithm was tested on a wide variety of geometries and was proven to detect the holes in discrete surfaces with excellent accuracy. These algorithms can also be repurposed to compute the genus of discrete surfaces. In many cases, the number of pairs of holes is the Surface's genus. However, this may not be true in scenarios where the algorithm overpredicts the holes. The algorithm relies heavily on half-edge data structure and feature angles and Gaussian curvature. For surfaces with connectivity, these are relatively easy metrics to compute. However, discrete surfaces represented in terms of point clouds do not have any pre-established connectivity, and hence, these algorithms are not applicable to these groups of surfaces. In the future, we plan on introducing a variant of the algorithm that works for point clouds by taking advantage of the progress made in point cloud processing.

## Acknowledgment

Österreichische Forschungsförderungsgesellschaft has funded this research under an industrial Ph.D. grant titled "HIOMESH."

## Conflict of Interest

There are no conflicts of interest.

## Data Availability Statement

The datasets generated and supporting the findings of this article are obtainable from the corresponding author upon reasonable request. The data and information that support the findings of this

article are freely available at: https://dx.doi.org/10.17632/3snr7g9cvx.1

## Nomenclature

$V$ = vertex of a mesh
$K_v$ = discrete Gaussian curvature
$\theta_i$ = incident angle for a given vertex

## Appendix A: Exhaustive Brute Force Search Algorithm for Solids

This algorithm searches across a very exhaustive search space and is the most naive way to find holes in discrete surfaces. The runtime of this algorithm is inferior and is very slow compared to any other algorithm discussed in the article. It is not liable for any practical CAD applications, and there is no way to improve its runtime. Any tuning should be done by tweaking the feature angle for search or with different curvature computation algorithms. We are also exploring possibilities to compute curvature on noisy surfaces by adding some additional threshold.

**Algorithm 4** Algorithm to detect holes in solids (naive exhaustive search algorithm)

**Result**: Vector of Chains (Every chain is a closed loop of edges)
Initialize Surface into a half-edge data structure;
Create an Empty vector called **possible_edges** to store edges;
**forall** *Faces of Surface* **do**
    Get the three surrounding faces;
    Compute the feature angle between the current face and its neighbors;
    **if** *The feature angle matches a specific feature angle criterion* **then**
        add them to possible_edges vector;
    **end**
**end**
Loop through **possible_edges vector** and form chains;

## Appendix B: Additional Experiments

We mentioned in this article that our algorithm succeeds in detecting most holes. Table 1 presents the number of holes detected by the algorithm for different geometries.

There are some cases where Gaussian-filtered search or even brute force algorithm would fail to detect holes due to poor triangulation. Scanned geometries also lack continuity in curvature and feature angles. Even sophisticated segmentation algorithms fail to detect any holes in such cases. It can be seen that a simple geometry in Fig. 5 has protruding triangles that are not planar. In such cases, the feature angle range could be increased, or the geometry needs to be repaired to avoid/remove such protrusions. In the case of industrial geometries, these are quite common and are not easy to detect and fix automatically. These geometries need to be preprocessed in a commercial CAD clean up tool for removing such defects in the geometry. There are also some cases where the algorithm overpredicts holes due to their reliance on feature angles. The unnecessary holes can be deleted interactively by the end-user.

**Table 1 Comparison of actual holes in the geometry vs. holes detected by the algorithm**

| Geometry | Actual holes | Detected holes |
|---|---|---|
| Cuboid with holes | 2 | 2 |
| Door Latch | 24 | 31 |
| Multiple cuboids with holes | 124 | 124 |
| Cylindrical sheet | 2 | 2 |
| Fidget spinner | 10 | 10 |
| L bracket | 4 | 6 |
| Camera mounting bracket | 6 | 6 |
| Geometry with cylindrical hole | 2 | 2 |
| Twin mounts | 4 | 4 |
| Cuboid with one hole | 2 | 2 |
| Gear with holes | 3 | 3 |
| Slab with multiple holes | 112 | 112 |

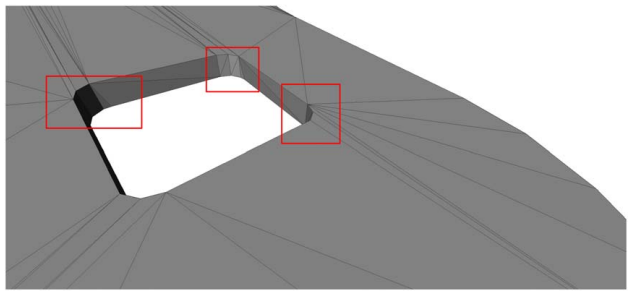

**Fig. 5 A hole with protruding faces**

## References


[1] Danglade, F., 2015, "Estimation of CAD Model Simplification Impact on CFD Analysis Using Machine Learning Techniques," CAD'15, London.
[2] Huang, J., Su, H., and Guibas, L. J., 2018, "Robust Watertight Manifold Surface Generation Method for Shapenet Models," arXiv preprint.
[3] Doraiswamy, H., and Natarajan, V., 2009, "Efficient Algorithms for Computing Reeb Graphs," Comput. Geom., **42**(6), pp. 606–616.
[4] Lozano-Durán, A., and Borrell, G., 2016, "Algorithm 964: An Efficient Algorithm to Compute the Genus of Discrete Surfaces and Applications to Turbulent Flows," ACM Trans. Math. Softw., **42**(4), pp. 34:1–34:19.
[5] Smereka, M., and Duleba, I., 2008, "Circular Object Detection Using a Modified Hough Transform," Int. J. Appl. Math. Comput. Sci., **18**(1), pp. 85–91.
[6] Wang, Y., Liu, R., Li, F., Endo, S., Baba, T., and Uehara, Y., 2012, "An Effective Hole Detection Method for 3d Models," 2012 Proceedings of the 20th European Signal Processing Conference (EUSIPCO), Bucharest, Romania, pp. 1940–1944.
[7] Joy, K. I., Legakis, J., and MacCracken, R., 2003, "Data Structures for Multiresolution Representation of Unstructured Meshes," *Hierarchical and Geometrical Methods in Scientific Visualization*, Farin, G., Hamann, B., and Hagen, H., eds., Springer, Berlin, Heidelberg, pp. 143–170.
[8] Botsch, M., Steinberg, S., Bischoff, S., Kobbelt, L., and Aachen, R., 2002, Openmesh—A Generic and Efficient Polygon Mesh Data Structure, RWTH Aachen.
[9] The CGAL Project, 2019. *CGAL User and Reference Manual*, 4.14 ed., CGAL Editorial Board.
[10] Meyer, M., Desbrun, M., Schröder, P., and Barr, A. H., 2003, "Discrete Differential-Geometry Operators for Triangulated 2-Manifolds," *Visualization and Mathematics III*, Hege, H.-C, and Polthier, K., eds., Springer, Berlin, Heidelberg, pp. 35–57.
[11] Long, J., 2013, "A Hybrid Hole-Filling Algorithm," Master's thesis, Queen's University, Kingston, Ontario, Canada.
[12] Pérez, E., Salamanca, S., Merchán, P., and Adán, A., 2016, "A Comparison of Hole-Filling Methods in 3d," Int. J. Appl. Math. Comput. Sci., **26**(4), pp. 885–903.
[13] Gärtner, B., 1999, "Fast and Robust Smallest Enclosing Balls," Proceedings of the 7th Annual European Symposium on Algorithms, Prague, Czech Republic, Springer-Verlag, pp. 325–338.