Formalizing Linear Motion G-Code for Invariant Checking and Differential Testing of Fabrication Tools

YUMENG HE, University of Utah, USA
CHANDRAKANA NANDI, Certora Inc. and University of Washington, USA
SREEPATHI PAI, University of Rochester, USA

The computational fabrication pipeline for 3D printing is much like a compiler — users design models in Computer Aided Design (CAD) tools that are lowered to polygon meshes to be ultimately compiled to machine code by 3D slicers. For traditional compilers and programming languages, techniques for checking program invariants are well-established. Similarly, methods like differential testing are frequently used to uncover bugs in compilers themselves, which makes them more reliable.

The fabrication pipeline would benefit from similar techniques but traditional approaches do not directly apply to the representations used in this domain. Unlike traditional programs, 3D models exist both as geometric objects (a CAD model or a polygon mesh) as well as machine code that ultimately runs on the hardware. The machine code, like in traditional compiling, is affected by many factors like the model, the slicer being used, and numerous user-configurable parameters that control the slicing process.

In this work, we propose a new algorithm for lifting G-code (a common language used in many fabrication pipelines) by denoting a G-code program to a set of cuboids, and then defining an approximate point cloud representation for efficiently operating on these cuboids. Our algorithm opens up new opportunities: we show three use cases that demonstrate how it enables (1) error localization in CAD models through invariant checking, (2) quantitative comparisons between slicers, and (3) evaluating the efficacy of mesh repair tools. We present a prototype implementation of our algorithm in a tool, GlitchFinder, and evaluate it on 58 real-world CAD models. Our results show that GlitchFinder is particularly effective in identifying slicing issues due to small features, can highlight differences in how popular slicers (Cura and PrusaSlicer) slice the same model, and can identify cases where mesh repair tools (MeshLab and Meshmixer) introduce new errors during repair.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Theory of computation → Denotational semantics**; **Operational semantics**; • **Applied computing → Computer-aided manufacturing**.

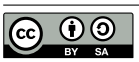
Additional Key Words and Phrases: G-code, operational semantics, invariant checking, differential testing

1 Introduction

Programming language and compiler researchers have noted similarities between traditional compiler toolchains and computational fabrication pipelines. Akin to source code, designers prepare a 3D model in computer-aided design (CAD) programs, export it to a polygon mesh which functions like a conventional intermediate representation (IR), that is then "compiled" to instructions for a computer-aided manufacturing (CAM) tool which runs these instructions to fabricate the physical model. Over the years, there has been work on correct-by-construction computer-aided design (CAD) compilers, tools that reverse engineer polygonal meshes to CAD models, program synthesizers for CAD, and domain specific languages (DSLs) targeting various fabrication related tasks [29, 72, 73, 87, 96, 101–103].

In this work, we use techniques from program semantics and differential testing to automate the detection of errors in models as well as in tools used in the 3D printing fabrication pipeline.

Authors' Contact Information: Yumeng He, u1528477@umail.utah.edu, University of Utah, Salt Lake City, UT, USA; Chandrakana Nandi, cnandi@cs.washington.edu, Certora Inc. and University of Washington, Seattle, WA, USA; Sreepathi Pai, sree@cs.rochester.edu, University of Rochester, Rochester, NY, USA.



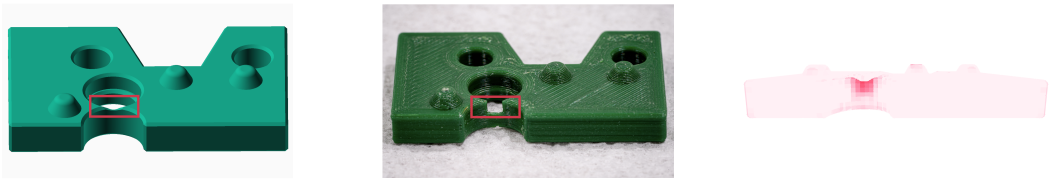


Fig. 1. 3D printed mechanical part (center) diverges from the original design (left) due to thin walls. The slightly curved gap in the model (highlighted by the red square) when printed, appears as a rectangular gap. The heatmap generated by our work (right) reveals this deviation before the model is printed.

The 3D printing fabrication pipeline. First, a user designs their model using geometric constructs in a CAD software. These can be textual programmatic systems like OpenSCAD [77], or interactive 3D geometric manipulation software like SolidWorks [92], Onshape [76], and Rhino [83]. The CAD software then generates a low-level polygon mesh by compiling the user's high-level design into a set of triangles that represent the surface of the model. Next, a *slicer* (e.g., PrusaSlicer [81], Cura [23], Simplify3D [88], Slic3r [89]) slices the mesh, usually to generate 2D horizontal layers from which it then emits G-code [55, 90] for a 3D printer to execute.

G-code is a programming language for numerical control, developed in the 1960s and defined by the RS-274 standard [55]. It is the de facto standard for computer aided manufacturing on machines like 3D printers, laser cutters, and CNC (Computer Numerical Control) mills. G-code is analogous to low-level machine instructions but omits constructs like loops and conditionals. The 3D printers we focus on are fused-deposition modeling (FDM) printers, which heat and melt polymer filament and lay it down on a "print-bed" through an extruder to create a 3D object. The process of laying down the plastic is specified by commands embedded in the G-code that move the extruder in 3D space while extruding plastic. The 3D printer's firmware [54, 65] interprets these commands. Other G-code commands heat the print bed and the extruder, select and switch between possibly multiple extruders, perform calibration, etc. [1].

Challenges with 3D printing. In practice, 3D printing requires repeated attempts before a successful print [46]. A part may fail to print correctly due to various reasons. The design itself may have issues like feature sizes that are too small for a given hardware. The mesh may have problems like flipped normals due to which the "inside" and "outside" of the model may be swapped. And finally, the slicer may fail to generate G-code due to bugs in the slicer itself or in some cases the generated G-code may not accurately represent the original model. To overcome these challenges, users may have to change/fix the design, fix mesh errors (e.g., holes, duplicate vertices and faces, invalid 2-manifold) using (often multiple!) mesh repair tools like MeshLab [110] and Meshmixer [6] before slicing, adjust various settings that are used to configure slicers like filament extrusion rate, temperature, change filament, and so on, or use different slicers to overcome slicing errors and differences due to heuristics used in slicing algorithms. Since 3D printers are slow, a single 3D print can take from many hours to days, making this iterative process time and resource intensive.

Debugging G-code is like debugging low-level assembly. Originally intended for small, low-cost microcontrollers optimizing for factors like frame resonance, acceleration, and toolhead inertia, standard G-code lacks control flow and uses fully unrolled loops. This results in extremely long, straight-line programs. Commands specify absolute or relative positions via floating-point constants derived from high-level 3D models, making it hard to trace code back to the original geometry.

The Problem: Lack of G-code analysis tools. Being able to (1) check invariants of the G-code before starting a print, and (2) compare the output of various tools that are used for slicing and repairing meshes via differential testing can help make the G-code more reliable and the tools

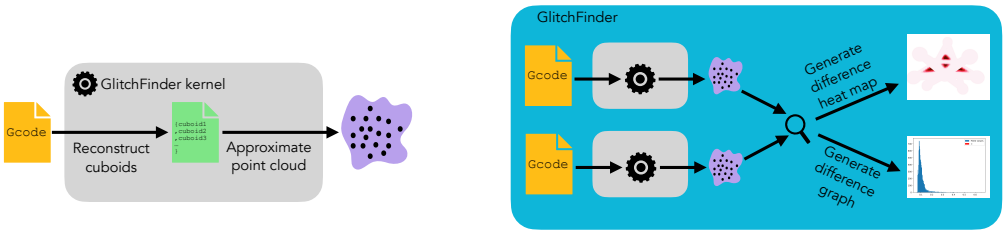


Fig. 2. (Left) GlitchFinder's kernel first reconstructs a set of cuboids from a G-code program to denote it. From the reconstructed cuboids, GlitchFinder then generates a point cloud. (Right) This kernel can then be used for comparing two G-code programs to generate a difference heatmap and a difference distribution graph. As later sections will show, both model invariant checking and differential testing can be reduced to a comparison of two G-code programs.

more robust, ultimately leading to a more efficient fabrication process. In traditional compilers and programming languages, there is decades of research on program analysis and differential testing; similar techniques would benefit computational fabrication.

However, traditional approaches do not directly apply to this domain because representations like CAD designs, polygon meshes, and G-code are different from typical programs. CAD designs and meshes are for example representations of an object's geometry, whereas the G-code that ultimately runs on CNC machines is expected to produce the *same* geometry (modulo physical constraints) while also being affected by factors like the slicer being used, the printer the G-code is being generated for, and the settings used to slice the model [95]. Two G-code programs that correspond to the same model can also be different based on the slicing algorithm used, making it even harder to compare G-code programs from two different slicers.

The choice of representation being analyzed or used for comparing slicers and mesh repair tools is an important one — a model may represent a valid geometry at the CAD or mesh level but fail to slice or print correctly (Figure 1 shows a real-world example). Thus using a mesh representation for differential testing of slicers and mesh repair tools may reveal little about whether the output of these tools will ultimately lead to successful printing and slicing.

Our solution and insights. We address these challenges by proposing a new technique for analyzing G-code programs. Our work is guided by the following three key insights.

Our *first* insight is that analyzing G-code captures the combined effect of the CAD design, the slicer used, and slicer settings. This is analogous to analyzing low-level bytecode as opposed to source code. We show how this "analyze-what-executes" approach helps check key invariants and localize potentially problematic regions of a model that may not print successfully, even if the model at the CAD design or mesh level represents a valid geometry.

Our *second* insight is a G-code program can be lifted by constructing a set of cuboids that *denotes* the program. Being able to represent a G-code program in this manner gives us a way to analyze G-code programs and compare them by comparing the set of cuboids they represent.

Our *third* insight is that by comparing two G-code programs we can differentially test mesh repair tools and slicers. We show that this is an effective approach for comparing the behavior of different slicers, and for evaluating the efficacy of popular mesh repair tools.

As shown in the heatmap in Figure 1 (right), our technique successfully localizes the part of the model in Figure 1 (left) that failed to print correctly (dark red in Figure 1, center). Crucially, it finds the problem by statically analyzing the G-code, *before* the model is printed.

We built a prototype tool, GlitchFinder, to implement our approach. GlitchFinder reconstructs a set of cuboids that *denotes* a G-code program (Figure 2, left). To efficiently compare two G-code programs, GlitchFinder uses a sampling-based approach to approximate a point cloud from the cuboid set (Figure 2, left). GlitchFinder uses a second algorithm to efficiently compare two point clouds to generate a difference heatmap for visualizing the differences between two programs (Figure 2, right). We use GlitchFinder to (1) localize errors on 56 complex, real-world, 3D models, (2) compare two popular slicers to identify different behaviors on the same models, and, (3) compare two popular mesh repair tools to evaluate their efficacy. This paper makes the following contributions:

- A new method for analyzing G-code programs by reconstructing a cuboid set which is then approximated as a point cloud and an algorithm for comparing two point clouds.
- Using point cloud comparison for invariant checking to localize problematic parts of a 3D model that cannot be identified from the CAD design or mesh representation.
- Using point cloud comparison to differentially test slicers and mesh repair tools.
- A prototype implementation in a tool dubbed GlitchFinder, and its evaluation over real-world models, popular slicers, and mesh repair tools.

The rest of the paper is structured as follows: Section 2 provides background on slicing and G-code, Section 3 presents big step operational semantics for the linear motion subset of G-code this work targets and our cuboid reconstruction algorithm, Section 4 presents a sampling-based point cloud generation approach from the reconstructed cuboids and a new algorithm for comparing two G-code programs, Section 5 discusses showing the output of comparing two G-code programs, Section 6 has implementation details and limitations, Section 7 shows how G-code analysis helps check invariants of 3D models, Section 8 shows how G-code comparison enables differential testing of slicers and mesh repair tools, Section 9 presents related work, and Section 10 concludes.

2 Background on G-code

```
...
G1 X151.801 Y158.165
G1 F1800 E-0.75
G1 F600 Z2.3
G0 F18000 X150.411 Y157.446 Z2.3
G0 X159.094 Y155.912
G1 F600 Z2.1
G1 F720 E0.75
G1 F2400 X160.02 Y156.838 E0.147
G1 X159.914 Y156.974 E0.01936
...
```

Figure 3 shows three different representations of the mechanical part in Figure 1 alongside analogies to a traditional compiler pipeline. The high-level CAD design must first be converted into a triangle mesh representation from which a slicer (e.g., Cura [23]) generates G-code, a preview of which is shown to the right in Figure 3 (including a close up view of the extruded lines in the preview). A slicer must first be configured with a variety of settings before it can generate G-code [95] with the most important one being the model of the 3D printer for which the G-code is to be generated. This, in turn, dictates additional settings like the diameter, d, of the nozzle on the printer's extruder, from which material is deposited. This work focuses on G-code representing linear motion, generated through the commonly used approach of *uniform, planar slicing* [28, 60]. For a model of height H and for a uniform layer height h, this method generates N slices where $N = \lceil H/h \rceil$. The layer height h is another parameter that is set to a fixed value in the slicer before slicing a model. These slices are parallel to a horizontal "print bed" as shown in Figure 3 (right) — the extruder of a 3D printer deposits filament along the 2D lines in each layer.

The snippet above shows examples of G-code instructions from the preview shown in Figure 3 (right).[1] A single straight line may be broken down into multiple instructions depending on the specific slicing algorithm used in a slicer. **G0** represents movements that do not extrude filament,

[1]The full G-code program contains other non-motion commands for setting temperature, selecting from among multiple extruders, and controlling peripherals such as lights.

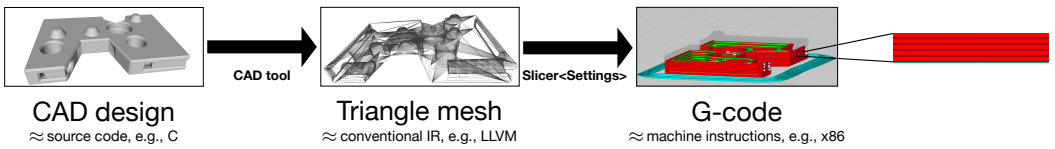


Fig. 3. Left to right: high-level CAD design, triangle mesh, a preview of the generated G-code for the mechanical part shown in Figure 1, and a zoomed in view of the cuboids represented by each G-code instruction which forms the basis of our semantics. Below each representation, we provide an analogy from a traditional compiler pipeline. The extruder of a 3D printer deposits filament along the horizontal lines in each layer of the G-code to manufacture the object.

whereas **G1** represents extruding movements. In the instruction, **G0** **F**18000 **X**150.411 **Y**157.446 **Z**2.3, **F** represents the "feed rate" which indicates how fast the extruder must move (18000mm/min), and **X**, **Y**, and **Z** indicate the absolute (or relative[2]) position of the extruder after this instruction is executed. For example, in the absolute setting, after executing this instruction the extruder would end up at the 3D coordinate: (150.411, 157.446, 2.3).

When an argument is omitted, its last value is used. Therefore, in the next instruction in the snippet, **G0** **X**159.094 **Y**155.912, the extruder moves to the coordinate (159.094, 155.912, 2.3) at the same speed as the previous instruction. Here, the **Z** coordinate also remains the same. The difference between any two consecutive distinct values of **Z** should always be h, the layer height chosen for this model (in this case, 0.2mm). The **E** in the instruction **G1** **F**2400 **X**160.02 **Y**156.838 **E**0.147 states that 0.147mm of filament is extruded during this movement. Commands like **G0** and **G1** are similar to assembly instructions and intended to run on CNC machines. G-code programs are *stateful*: both instructions update the position of the toolhead (e.g., extruder of a 3D printer) and **G1** additionally also adds filament either directly on the print bed or on top of previous layers. This observation guides how we define the state of a G-code program in Section 3.1.

3 Formalizing Linear Motion G-code Program

Our key idea is that by analyzing G-code programs, (1) we can check invariants of models and localize problematic parts, and (2) compare two G-code programs for differentially testing slicers and mesh repair tools. To analyze a G-code program comprised only of linear motion, we lift the program to a set of cuboids and approximate it as a point cloud. To that end, we first provide formal semantics for G-code.

3.1 Semantics of gcode_d^h that Guides Cuboid Reconstruction

We define gcode_d^h to be a G-code program that is parametrized by d, the extruder nozzle diameter on the 3D printer for which some slicer generated the G-code, and h the fixed layer height used when configuring the slicer. Figure 4 shows the formal syntax of gcode_d^h, specifically the subset of G-code this paper targets: linear motion G-code. In Figure 4, pos represents the position arguments of **G0** and **G1** and attribute is used for arguments **E** and **F**. A gcode_d^h can be empty or a list of commands (we model linear motion commands **G0** and **G1**).

These commands (cmd) are instructions for an extruder on a 3D printer to deposit filament in a 2D horizontal layer such that the resulting material (e.g., plastic) deposition constructs the 3D model. The extruder has a heating element that melts the material which comes out from a nozzle which typically has a round hole. If the nozzle was to extrude filament in free space, the filament

[2]G-code commands `G90` and `G91` switch between absolute and relative positioning respectively

$$
\begin{array}{lll}
\text{pos} & ::= & (\mathbb{R}, \mathbb{R}, \mathbb{R}) \qquad \text{len} ::= \mathbb{R} \qquad \text{rate} ::= \mathbb{R} \qquad \text{attribute} ::= \text{E len} \mid \text{F rate} \\
\text{cmd} & ::= & \text{G0 (pos, attribute*)} \mid \text{G1 (pos, attribute*)} \\
\text{gcode}_d^h & ::= & \text{empty} \mid \text{cmd} :: \text{gcode}_d^h
\end{array}
$$

Fig. 4. Syntax of the subset of G-code this paper targets: linear motion generated by 3D printing slicers. We define gcode_d^h to be a G-code program whose layer height set to h and which is generated for a 3D printer whose extruder nozzle has diameter d.

Fig. 5. (Left) Top view of the cuboid shown to the right. $r = d/2$ is the radius of the nozzle on an extruder that is moving from point A to B. The outer rectangle $CDEF$ approximates a rectangle of length AB together with two semicircles protruding from either end. Intuitively, imagine these rounded ends are made by a circle of radius r centered at A that is dragged to the point B. The dashed arrow depicts the trajectory of the nozzle's center — it moves from left to right. (Right) Reconstructed cuboid that denotes the line AB extruded by the purple nozzle whose diameter is d.

would therefore form a *cylindrical* shape along a line segment (similar to toothpaste out of a tube in air). However, typically the nozzle extrudes on a horizontal print bed (subsequent layers are extruded on top of the previous layer). The extruded cylinder will thus be "squished" between the nozzle and the print bed (or previous layer) such that the top and bottom faces of the extruded material are parallel to each other. Figure 3 (right) shows each extruded line up close for visual understanding. Thus, the horizontal cross-section of an extruded line of material is a flat rectangle with round corners.

Figure 5 (left) demonstrates this visually — AB represents the top view of an extruded line between two consecutive G-code instructions. The rounded ends can be imagined as being constructed by dragging a circle of radius r centered at A to the point B, where $r = d/2$ is the radius of the nozzle. We ignore the rounded corners and approximate the line segments with the outer cuboids (top view shown by $CDEF$ in Figure 5, left) whose height is set to be the uniform layer height h in the slicer when generating the G-code for the model. Figure 5 (right) shows the 3D view of the reconstructed cuboid whose top *face* is $CDEF$.

Figure 6 shows the big step operational semantics of gcode_d^h. We define the state σ of a gcode_d^h program to be a tuple whose first element, p_e, is the current position of the extruder nozzle in 3D space. We assume that initially it is at origin (see top of Figure 6). The second element in the tuple is a set of cuboids that denotes the 3D object being constructed by gcode_d^h. We represent a cuboid with its top face (f_t) and bottom face (f_b) where a face is a 4-tuple made from vertex as shown.

We define two judgments: $\Downarrow_{d,h}$ and $\Downarrow_{d,h}^{p}$. The first, $\Downarrow_{d,h}$, evaluates a cmd (as shown in Figure 4) to produce a new state. The attributes do not affect the semantics, as shown by _ in the inference rules for **G0** and **G1**. In the case of **G0**, no new cuboid is added in the new state: it remains cs which was the set of cuboids in the previous state. In the case of **G1**, a new cuboid is added to cs shown by $cs \cup \{(f_t, f_b)\}$. The vertices for f_b are computed as shown in Figure 6 using the rules of trigonometry. The variable k in Figure 6 represents the distance from a line's endpoint to its nearest cuboid vertex — in Figure 5 (left), k is the length of AC, AF, BD, and BE.

$$x,\ y,\ z \in \mathbb{R} \qquad \text{vertex} ::= (x,\ y,\ z) \qquad \text{origin} ::= (0, 0, 0)$$

$$\text{face} ::= (\text{vertex},\ \text{vertex}, \text{vertex},\ \text{vertex}) \qquad \text{cuboid} ::= (\text{face}_t,\ \text{face}_b)$$

$$\text{state}\ \sigma ::= (p_e,\ \{\ \text{c} \mid \text{c}\ \textit{is a}\ \text{cuboid}\ \}) \qquad p_e ::= \text{vertex}$$

$$\frac{\sigma;\ s\ \Downarrow_{d,h}\ \sigma' \qquad \sigma';\ ss\ \Downarrow^{p}_{d,h}\ \sigma''}{\sigma;\ s\ ::\ ss\ \Downarrow^{p}_{d,h}\ \sigma''}\ \text{gcode}^h_d \qquad \frac{}{(p_e,\ cs);\ \mathbf{G0}(p,\ \_)\ \Downarrow_{d,h}\ (p,\ cs)}\ \text{G0}$$

$$\frac{cs' = cs\ \cup\ \{(f_t,\ f_b)\}}{(p_e,\ cs);\ \mathbf{G1}(p,\ \_)\ \Downarrow_{d,h}\ (p,\ cs')}\ \text{G1} \quad \text{where}$$

$$\begin{aligned} f_b \quad = \quad & (p_e\ +_v\ (-k \cdot \cos(-\theta + \tfrac{\pi}{4}),\ k \cdot \sin(-\theta + \tfrac{\pi}{4}),\ p_{e_z}),\\ & p\ +_v\ (k \cdot \cos(\theta + \tfrac{\pi}{4}),\ k \cdot \sin(\theta + \tfrac{\pi}{4}),\ p_{e_z}),\\ & p\ +_v\ (k \cdot \cos(-\theta + \tfrac{\pi}{4}),\ -k \cdot \sin(-\theta + \tfrac{\pi}{4}),\ p_{e_z}),\\ & p_e\ +_v\ (-k \cdot \cos(\theta + \tfrac{\pi}{4}),\ -k \cdot \sin(\theta + \tfrac{\pi}{4}),\ p_{e_z})), \quad k = \sqrt{2}(d/2), \quad \theta = \tan^{-1}(slope(\overrightarrow{p_e p})) \end{aligned}$$

$$f_t \quad = \quad \text{map}(\lambda v.\ (v\ +_v\ (0,\ 0,\ h)),\ f_b)$$

Fig. 6. Big step operational semantics of gcode^h_d. From Figure 5 (Left), A is p_e, B is p, $CDEF$ is f_t, and $r = d/2$. We define $+_v$ as the addition operator over two 3D vertices. As is standard, map applies its first argument, a function, to all vertices in f_b. The z component of p_e is shown by p_{e_z}.

We use $+_v$ to indicate addition of two 3D vertices: $v_1 +_v v_2 = (v_{1x} + v_{2x}, v_{1y} + v_{2y}, v_{1z} + v_{2z})$. Here, θ is the gradient (computed using the helper function *slope* in Figure 6) of the line joining p_e and p, shown by $\overrightarrow{p_e p}$, i.e., θ is the angle between the x-axis and $\overrightarrow{p_e p}$.

Since the top and bottom face are parallel, the coordinates of the vertices of f_t only differ from those of f_b in the z coordinate (as we denote each instruction as a cuboid), which is offset by the layer height, h. For both **G0** and **G1**, p_e is updated to p which is the position argument. This indicates that after this command is executed, the new position of the nozzle is p.

The judgment $\Downarrow^{p}_{d,h}$ evaluates a gcode^h_d program as shown in Figure 6 by folding over all the commands in the program. The rule states that at state σ, if the command s evaluates to produce the σ' following the rules of $\Downarrow_{d,h}$, and at state σ', the rest of program, ss evaluates to σ'', then the entire program gcode^h_d evaluates to generate the final state σ''.

This explains how, given a gcode^h_d program, GlitchFinder reconstructs a set of cuboids that denotes the 3D model that would be printed upon executing gcode^h_d on a 3D printer (Figure 2, left).

3.2 Approximating gcode^h_d as a Point Cloud to Facilitate Comparison

Once a 3D model has been reconstructed from a gcode^h_d program as described in Section 3.1, it can be used for many downstream applications. This work demonstrates two: (1) invariant checking of 3D models, and (2) differential testing of mesh repair tools and slicers. Our key observation is that *both* tasks can be performed by *comparing* two gcode^h_d programs. This comparison process is non-trivial and naive approaches suffer from poor scalability, numerical errors, imprecise localization of differences, and hard-to-interpret results. This section and the ones that follow tackle each of these problems in turn.

Algorithm 1 Point Cloud Comparison Algorithm

```
procedure POINT_CLOUD_SEGMENTATION(pc1, pc2, ubox)
    bbox ← getBbox(pc1, pc2)
    n ← countUnitBox (ubox, bbox)
    boxed_point_sets ← [ [ [], [] ] × n]
    for all p ∈ P | P ∈ {pc1, pc2} do
        offset ← findBoxIndex (p, ubox, bbox)
        boxed_point_sets [offset][i].append (p)          ▷ i ∈ {0, 1} depends on if p ∈ pc1 or pc2
    end for
    return (boxed_point_sets, n)
end procedure
procedure COMPARE_POINT_CLOUDS(boxed_point_sets, n)
    hd_list ← []
    for all i ∈ range(n) do
        P_1 ← boxed_point_sets [i][0]
        P_2 ← boxed_point_sets [i][1]
        P_1' ← []                                        ▷ P_1's neighborhood
        P_2' ← []                                        ▷ P_2's neighborhood
        for all j ∈ neighboringBox (i) do
            P_1'.extend(boxed_point_sets[j][0])
            P_2'.extend(boxed_point_sets[j][1])
        end for
        hd_list.append(max (oneWayHD(P1, P2'), oneWayHD(P2, P1')))
    end for
    return hd_list
end procedure
```

A naive approach for comparing two gcode_d^h programs, p1 and p2, once a 3D model is reconstructed from gcode_d^h as a set of cuboids, is to use set-theoretic constructive solid geometry operations like *difference* [73, 77, 104] that are available in CAD tools: difference (union (c11, c12, ..., c1n), union (c21, c22, ..., c2m)), where union (c11, c12, ..., c1n) is the set of cuboids for p1, and union (c21, c22, ..., c2m) is the set of cuboids for p2. If the result is $\emptyset$, that means the two programs *denote* the same 3D solid. A nonempty result indicates differences. However, early experimentation showed that for large models this fails to scale.

Therefore, in GlitchFinder, we present a more practical algorithm for comparing two gcode_d^h programs. Our algorithm discretizes the cuboids into a set of points yielding a *point cloud* for the reconstructed model (Figure 2, left).

To generate a point cloud to approximate the reconstructed cuboid set, GlitchFinder samples the cuboids. A straightforward strategy is to use uniform sampling where the same number of points are sampled from each cuboid. However, this would oversample short cuboids and undersample long cuboids. We therefore define

$$\Omega_g : \text{cuboid} \rightarrow \text{points}$$

as a function that *proportionally samples* different numbers of points from differently sized cuboids, depending on its dimensions. We use the parameter g to indicate the *sampling gap*, which represents the distance between any pair of neighboring points. Ω_g then samples

$$(\lceil (l+d)/g \rceil + 1) \cdot (\lceil d/g \rceil + 1) \cdot (\lceil h/g \rceil + 1)$$

points for a cuboid with length $l + d$, where d is the nozzle diameter, l is the length of the extruded line, and h is the layer height. From Figure 5, the dimensions of the reconstructed cuboid are length: $(l + d)$, breadth: d, and height: h. This method produces point clouds that are usually smaller than

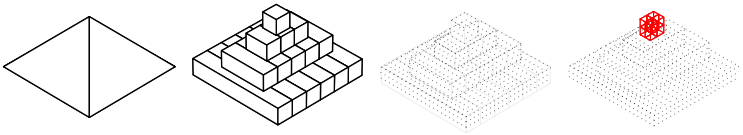


Fig. 7. Artifacts at different stages of processing: the input model whose G-code we are lifting, the reconstructed cuboid set, the approximated point cloud, and a subset of unit boxes overlaid on the point cloud.

what uniform sampling from the cuboids would produce, speeding up point cloud comparison. To summarize, if G is a gcode_d^h program and $G \Downarrow_{d,h}^{p} (\_, C)$, then $\text{flatmap}(\Omega_g, C)$ is the point cloud GlitchFinder generates for G. Next, we show how GlitchFinder compares two point clouds using an *augmented* Hausdorff distance metric.

4 Comparing Point Clouds

To compare two point clouds obtained as described in Section 3.2, we treat each point cloud as a *set* of points and introduce an augmented version of Hausdorff distance [71].

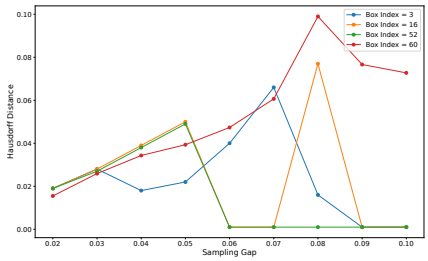
The Hausdorff distance is widely used for evaluating global similarity by calculating the maximum of nearest-neighbor distances between two point clouds [5, 41, 47]. However, because it yields a single metric for the entire comparison, we cannot localize faults to regions of the point cloud. To address this, GlitchFinder segments each point cloud into multiple *unit boxes* and performs a finer-grained analysis by comparing the subset of points within the corresponding unit boxes. This segmentation, done naively, causes wild swings in the Hausdorff metric as points on the boundaries of partitions get ascribed to different unit boxes due to floating-point errors. To counter this, we introduce a novel *augmented* Hausdorff metric that compensates for points displaced into adjacent unit boxes due to numerical errors.

4.1 Overall Approach

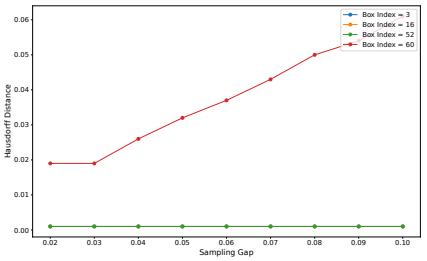
Our approach consists of two main steps: (1) point cloud segmentation followed by (2) point cloud comparison by Hausdorff distance computation within each segment. To perform point cloud segmentation, GlitchFinder first creates a single bounding box that encompasses the larger of the two point clouds to be compared (getBbox) if the extents of the point clouds are different. The bounding box is then uniformly divided into multiple smaller unit boxes. Users can specify the dimensions of a unit box, which typically depend on the scale of the smallest feature differences users aim to capture. We then place each point in each point cloud into a unit box depending on its spatial position. This procedure is shown in POINT_CLOUD_SEGMENTATION in Algorithm 1. The inputs to the procedure are the point clouds pc1 and pc2, and dimensions of the unit box, ubox. The function countUnitBox computes the total number of unit boxes based on the dimensions of the bounding box (bbox) and ubox. Then, the function findBoxIndex finds the closest unit box in which to place a point.

Now we can compare two point clouds by computing the distance between corresponding subsets of points in each unit box. Figure 7 visualizes the inputs and outputs of the different stages of reconstruction and comparison.

In our context, the distance can be a numerical value, none, or ∞, depending on the number of points in the unit box, u. When both point clouds contain points in u, the distance is a numerical value. If u is empty for both point clouds (i.e., neither point cloud has points in u), we define the the distance to be none. The ∞ case occurs when only one point cloud has points in u, while the



(a) Hausdorff distance for points in four random unit boxes under varying sampling gaps. As the sampling gap changes, the Hausdorff distance within all boxes fluctuates, showing a 10× change in value.



(b) Hausdorff distance within the same four boxes in 8a, obtained by using Equation 1. The Hausdorff distance now exhibits a monotonic, non-increasing trend as the sampling gap decreases.

Fig. 8. Hausdorff distance (y-axis) vs sampling gap (x-axis) with and without floating point error handling.

other does not. However, we observe that many such cases arise as side effects of discretization and quantization [32, 79] or floating-point errors.

To avoid labeling these cases as having ∞ distance and to better highlight the primary differences in model features, we adopt the following approach: for the point cloud that has no points in u, we construct a *neighborhood* of points centered around u. This neighborhood includes points from all adjacent boxes, i.e., boxes that either share a face, vertex, or edge with u. Including u, there are 27 such boxes. We then compute the Hausdorff distance from this neighborhood to the subset of points in u from the other point cloud. This is explained in detail in the following section (Section 4.2).

4.2 Handling Floating-point Error

When comparing two very similar point clouds, one would expect the Hausdorff distances to be relatively small if there is no significant difference in model features. Additionally, the distance should decrease as the sampling gap becomes smaller, since the sampled point cloud more closely resembles the original model. However, this expected behavior is not observed when the standard Hausdorff distance metric is applied to points within each unit box. To understand this phenomenon better, we compared two point clouds, both reconstructed from the gcode_d^h program of a simple rectangular prism model. One was the point cloud of the original model, and the other was the point cloud of a rotated version of the model which was rotated back after reconstruction. Ideally, these should be the same point cloud.

Figure 8a shows the Hausdorff distance between points in the two models within four randomly selected, nonempty unit boxes. At the smallest sampling gap (0.02), the Hausdorff distance is 10× higher than the distance at the largest sampling gap (0.10). We discovered that these non-monotonic fluctuations are caused by floating-point errors during point cloud segmentation. When findBoxIndex determines the unit box a point belongs to, imprecise floating-point arithmetic can result in a point from box A being incorrectly assigned to a different box B in the other model. Since Hausdorff distance [11, 71] takes the maximum distance from any point in one point set to the other point set, it is highly sensitive to inaccuracies arising from missing points in one of the sets. Therefore, we develop the following technique to mitigate the effect of floating-point errors in POINT_CLOUD_SEGMENTATION.

We present a new augmented Hausdorff distance formula as follows: the distance between two point sets X and Y in a unit box B is defined as

$$d'_H(X,\ Y)\ =\ \max\{\sup_{x\in X} d(x,\ Y'),\ \sup_{y\in Y} d(X',\ y)\} \tag{1}$$

where d denotes the standard Hausdorff distance formula between a point and a set of points. X' and Y' are the respective *neighborhoods* of X and Y (defined in Section 4.1), where a neighborhood of a point set within a unit box B contains points from all neighboring boxes of B, including points from B itself. This definition of $d'_H(X,\ Y)$ ensures that, even if a point x in X cannot find its closest point y in Y due to misclassification of y, x can still match with y in Y's neighborhood Y'. This procedure is shown in COMPARE_POINT_CLOUDS in Algorithm 1. It takes the segmented point clouds boxed_point_sets and number of unit boxes n as input and returns a list of augmented Hausdorff distances, hd_list. In the algorithm, oneWayHD(X,Y) is equal to $\sup_{x\in X} d(x,\ Y)$. The Hausdorff distance computed with Equation 1 now exhibits the expected monotonic behavior (Figure 8b).

5 Visualizing True Differences

Reconstructing point clouds from two gcode_d^h programs enables invariant checking and differential testing of fabrication tools by allowing gcode_d^h comparison. So far, we have described how GlitchFinder performs this reconstruction and comparison. We now discuss how the results are presented to the user. We visualize distances per unit box as a heatmap. However, mapping magnitudes linearly to colors is ineffective because, alongside the *true* differences caused by faulty models or slicer/mesh tool variations, there are also *unwanted* errors from discretization and quantization [23, 79] that are either inherent to slicing, and can be introduced during point cloud reconstruction, as well as floating-point errors inevitable in finite-precision arithmetic. Ideally, GlitchFinder should highlight only true differences while filtering out these inherent artifacts.

5.1 Spatial Averaging to Reduce Unwanted Errors

On examining the distribution of distances computed using Algorithm 1, we observed that discretization and quantization errors are typically confined to specific regions of the model, such as boundaries and areas connecting the inner wall and infill, while floating-point errors appear relatively randomly. Therefore, GlitchFinder averages distances spatially by computing the mean distance within each unit box and its immediate neighborhood. Specifically, the distance for unit box B is now calculated as $\left[\sum_{i=1}^{n} d_i\right]/n$, where n is the number of neighboring boxes of B that contain a numerical distance value. We preserve all distances that are none or ∞ without modification.

While this averages out the effect of random unwanted errors in the heatmap, it can also reduce the distances representing true differences. To address this, using box size smaller than the size of the feature difference that user wants to detect is preferable. As demonstrated in our evaluation (Table 8, Table 9 in Section 11), with sufficiently small unit box sizes, the heatmaps consistently highlight areas with true differences.

5.2 Choosing Colors for Visualization

Heatmap colors are drawn from a continuous spectrum. Points in each nonempty unit box share one color. Nonempty boxes that have ∞ distance are assigned the darkest possible color in the spectrum. Otherwise, users set a *threshold percentile* parameter to determine the cut-off value for distinguishing distances representing true differences from those representing unwanted differences. For example, a 90^{th} threshold percentile means that all distances smaller than the 90^{th} percentile of the distances are considered difference-free and are assigned the lightest possible color. This is used to highlight true differences by filtering out other unwanted errors. We devised this strategy

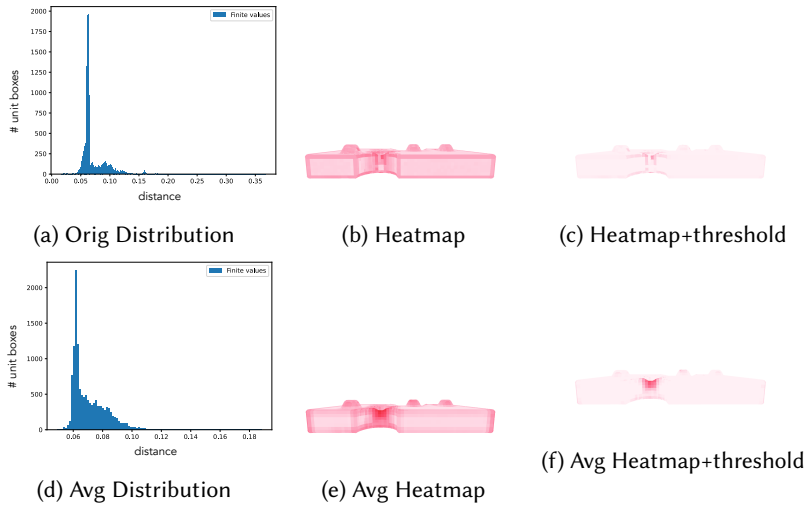


Fig. 9. Heatmap and distance distribution visualization: the bottom row uses spatial averaging.

after examining the distribution of true errors in detail, as described in Section 5.3. For distances between the defined threshold and ∞, we normalize their difference relative to the threshold over the entire distance range and assign colors based on the normalized distances.

5.3 Analyzing Trends in Distance Distributions

Across benchmarks, both raw (Algorithm 1) and spatially averaged distance distributions (Section 5.1) show that most unwanted errors, caused by discretization, quantization, or floating-point arithmetic, tend to have small magnitudes. True differences, on the other hand, are relatively infrequent compared to unwanted errors and typically exhibit much larger magnitudes. A distribution graph of distances between two models with notable feature differences often exhibits a right-skewed shape, where the maximum distance is significantly larger than the minimum. A high percentile threshold therefore effectively separates true differences from noise. In contrast, a distribution graph between two models *without* significant feature differences often tends to exhibit a nearly normal or slightly left-skewed distribution shape. The distribution may become more right-skewed as the unit box size decreases though still within a narrow range relative to the sampling gap. GlitchFinder visualizes these differences via both heatmaps and distribution graphs to help distinguish true faults. Figure 9 visualizes the difference heatmap for the model in Figure 1. The final result (bottom right) demonstrates how averaging and using a threshold (90^{th} percentile) allows immediate identification of the fault in the model.

6 Implementation

We implemented GlitchFinder in Python in approximately 1,200 LOC and it is publicly available [43]. The GlitchFinder "kernel" takes as input a G-code file (with `.gcode` extension) and following the semantics in Section 3, reconstructs a set of cuboids. From that, the kernel generates an approximate point cloud as explained in Section 4. The kernel can be used to compare two G-code programs and output a heatmap image and a distance distribution graph. This can be used for various downstream applications; this work demonstrates two (Section 7, Section 8).

6.1 Complexity

We assess GlitchFinder 's scalability by first deriving the asymptotic cost of each of its components and showing that every stage grows linearly with the input geometry. Section 7.4 provides running times from our evaluation.

GlitchFinder parses the input G-code files; for each linear motion command (with extrusion), it computes vertices of the corresponding cuboid, making this phase run in time linear in the number of commands. The proportional sampler is also linear in the number points generated. If the two input models are misaligned, we rotate the second point cloud; this operation touches each of its points only once and therefore runs in time proportional to the number of points in that cloud. In point cloud segmentation (Algorithm 1), the only non-constant work is the loop that assigns each point to its corresponding unit box, so the running time is linear in the total number of points across the two clouds. For point cloud comparison we iterate over the unit boxes; in each box we compute an augmented Hausdorff distance between the two point subsets it contains using SciPy's[3] `scipy.spatial.KDTree`: building a tree for a subset of m points and querying it once per point costs $O(m \log m)$, yielding an overall $O(N \log N)$ across all points N. The pipeline finishes with error distribution and heatmap generation, each running in time linear in the number of unit boxes.

In general, point cloud comparison is the most time-consuming phase, with runtime dominated by SciPy's `KDTree` operations. Faster nearest-neighbor structures or optimized libraries could reduce this time. The rest of the runtime is largely I/O. Additional implementation-level optimizations are also possible: for example in the invariant checking application (Section 7), several comparisons share the same original G-code; parsing and sampling that file once and reusing the resulting point cloud across all comparisons would eliminate redundant work.

6.2 Limitations

GlitchFinder's cuboid reconstruction algorithm uses a rectangular cuboid as an approximation for each deposited plastic segment, which may introduce inaccuracies in modeling the G-code. In particular, this prevents GlitchFinder from modeling G-code programs that use non-planar or non-uniform slicing. G-code commands that encode non-linear movement (e.g., circular motion) are also not supported; we only model **G0** and **G1**. We produce a logical approximation of the actual 3D model and do not consider the influence of temperature variations or filament properties. GlitchFinder does not model friction, vibration, or mechanical properties, and focuses only on the "core" program that is responsible for constructing the model ignoring non-motion instructions that are present in all G-code programs. This means that GlitchFinder targets errors that are directly attributable to digital artifacts: 3D designs, meshes, slicers, or a combination thereof. There may however be other kinds of errors that can be attributed to the hardware, like loose printer belts, insufficient (or excessive) cooling, etc. that GlitchFinder cannot identify.

7 Invariant Checking for 3D Models

This section evaluates our first claim that by analyzing gcode_d^h program using GlitchFinder, we can statically check invariants of models to detect regions that are likely to print incorrectly, even if they are valid 3D solids. First, we define a specific invariant of gcode_d^h programs checking which has helped us identify problems in many benchmarks.

Motivation. In traditional compilers, the idea of equivalence modulo inputs [57] has been used for testing compilers by perturbing programs to obtain semantically equivalent, yet syntactically different inputs. This idea can be instantiated for the 3D printing domain as well. For example, rotating a model should not affect the shape represented by the generated G-code; it should only

[3]https://scipy.org/

change its orientation in 3D space. In fact, it is common knowledge [3, 66, 121] among users interacting with 3D printing tools that comparing G-code programs sliced in different orientations can reveal problems in 3D models and even expose bugs in slicers.[4] Inspired by these observations, we define the following invariant for G-code programs.

Rotation Invariant, $\mathcal{I}$. We define a 3D triangle mesh as $(\mathbb{R}^3, \mathbb{R}^3, \mathbb{R}^3)$. Let R be a function that rotates a mesh by a 3D vector and slices it (by invoking a slicer) to produce gcode_d^h. Let r_c be a function that rotates a cuboid set by a 3D vector. Let r_p be a function that rotates a point cloud by a 3D vector.

$$\begin{array}{lll} R & : (\mathbb{R},\ \mathbb{R},\ \mathbb{R}) \rightarrow \text{mesh} & \rightarrow \text{gcode}_d^h \\ r_c & : (\mathbb{R},\ \mathbb{R},\ \mathbb{R}) \rightarrow \text{cuboids} & \rightarrow \text{cuboids} \\ r_p & : (\mathbb{R},\ \mathbb{R},\ \mathbb{R}) \rightarrow \text{points} & \rightarrow \text{points} \end{array}$$

Let M be a mesh that produces a gcode_d^h program G; let $G \Downarrow_{d,h}^{p} (\_, C)$ and let $P = \text{flatmap}(\Omega_g, C)$ be the point cloud approximated from C. Let G_v be the gcode_d^h obtained by slicing the rotated mesh, $R(v, M)$; let $G_v \Downarrow_{d,h}^{p} (\_, C_v)$, and let $P_v = \text{flatmap}(\Omega_g, C_v)$ be the point cloud approximated from C_v. The rotation invariant states:

$$r_c(v, C) = C_v$$

i.e., rotation and denotation must commute. However, as mentioned in Section 3.2, checking this does not scale to complex models. We therefore define and check the following approximate version of this invariant over the point clouds:[5]

$$r_p(v, P) \overset{\epsilon}{\approx} P_v$$

Observe that checking $\mathcal{I}$ has now reduced to comparing two gcode_d^h programs by comparing their point clouds which GlitchFinder enables.

Other affine transformations like translation and scaling also define valid invariants, i.e., they commute with denotation. For example, translating a 3D mesh (M) along the x-axis, slicing it, computing the denoted cuboid set, and then translating the cuboids back should yield the original cuboid set corresponding to M. The same holds for scaling and rotation. However, we hypothesize that $\mathcal{I}$ is more effective for revealing model bugs and aiding differential testing of slicers and mesh repair tools. Unlike translation and scaling which involve simple arithmetic, rotation requires trigonometric functions which empirically appear to expose more errors.

The rest of this section evaluates our claim that by checking $\mathcal{I}$, GlitchFinder can localize problematic parts in large, real-world 3D models for which we analyze the difference heatmap and distance distribution graph (Figure 2, right) produced by GlitchFinder.

7.1 Benchmarks

We used GlitchFinder to check $\mathcal{I}$ for 56 real-world models, including 50 problematic models that failed to produce correct G-code programs as reported by users, and 6 error-free models. Among the 50 problematic models, one is shown in Figure 1, a mechanical part from an open-source project [33]. To select the remaining models, we manually examined GitHub issues and user forums from two popular slicers, UltiMaker Cura [23][6] and Prusa [81]. We selected 50 issues / posts that included a model that we could use for our evaluation. We obtained the 6 error-free models by inspecting the Voron-0 3D printer parts repository [117] because parts for building 3D printers are more likely to be error-free due to widespread use and extensive testing.

[4] https://github.com/Ultimaker/Cura/issues/16726#issuecomment-1719546101

[5] We emphasize that this does *not* mean the 3D model must be physically *printed* in the rotated orientation; indeed many models are designed such that they can only successfully be printed in a particular orientation (e.g., due to overhang).

[6] We used the "Slicing Error" filter on Github.

7.2 Experimental Setup

Our GlitchFinder results were obtained on CloudLab [30] `xl170` machines which are 10-core Intel E5-2640v4 running at 2.4 GHz with 64GB RAM and 6G SATA SSDs. To generate G-code, we used the popular and actively maintained Cura slicer [23] running on Linux. We configure the slicer with the following settings: 0.4mm nozzle diameter, 0.2mm layer height 100% infill density, and no support structures or build plate adhesion (all other settings remained had default values).[7]

The axes and angles of rotation are parameters to GlitchFinder and therefore user configurable. By inspecting problematic models, we observed that rotations about the major axes often lead to different behaviors in problematic models. Since the slices are parallel to the XY plane (Section 2), we found that most of the interesting errors are exposed when the rotation changes the orientation of the object such that its base is no longer "on" the XY plane. As a result, rotation about the z-axis does not expose as many errors. Therefore, we chose to rotate each model by two different vectors: by $(90, 0, 0)$ and by $(0, 90, 0)$ (i.e., rotation about x and y axes). We also found that rotating by other angles like 45 degrees tends to increase unwanted errors (like quantization errors) that do not correspond to true differences.

For streamlining our evaluation, we wrote an automation script (called GlitchRunner) to which we pass the 3D triangle mesh from the original model, the rotation angles, the sampling gap (g), the unit box dimensions (see Algorithm 1 for point cloud segmentation), and a threshold percentile (see Section 5 for heatmap and distribution graph). These arguments are all user configurable. Section 11 lists the parameters for all our benchmarks. GlitchRunner automates the rotation, the slicing (by invoking the Cura slicer's `CuraEngine` directly), and ultimately invokes GlitchFinder. For some models, GlitchRunner first scales them to fit within Cura's build plate for before slicing. GlitchFinder's kernel (Figure 2, left) reconstructs the cuboid set and approximates the point cloud for each model, performs POINT_CLOUD_SEGMENTATION, and ultimately runs COMPARE_POINT_CLOUDS (Algorithm 1). Since each model is rotated both about the x and y axes, GlitchFinder produces two heatmaps and distribution graphs for each benchmark. GlitchRunner takes these and generates a combined heatmap and distribution graph by computing the average distance for each unit box.[8]

7.3 Results

In all 50 problematic benchmarks, GlitchFinder successfully identified the problematic areas. Based on the original discussion about the model posted by users, we classify these models into three categories based on the kind of problem they exhibit: models with small feature sizes (12), non-watertight models (33), and models with flipped normals (5), and discuss our findings.

7.3.1 Models with Small Features. Figure 10 presents the overall heatmaps and distribution graphs for a collection of 12 models with small geometries. GlitchFinder is particularly effective in these scenarios, as the differences in G-code resulting from small features are often too subtle and difficult to detect manually. Cura failed to slice the `Amy` and `Warrior` models entirely in their original orientations. To address this, we applied a 90-degree rotation about the z-axis and treated this rotated orientation as the original for these models.

We highlight four models (`Arm`, `VertexProblem`, `Dragon`, `Amy`): these exhibit varying degrees of mesh flaws, which are detectable by existing mesh tools according to the discussions in the Github issue where we found the models. However, in all these cases, the only flaw in the generated

[7]These settings are not intended for actual printing; they are only used for statically comparing two G-code programs using GlitchFinder.

[8]This requires that all hd_list are generated using the same dimensions of ubox and bounding box. During the averaging, distances with a none value are ignored, and whenever any distance within a unit box is ∞, the average distance for that box is set to ∞.

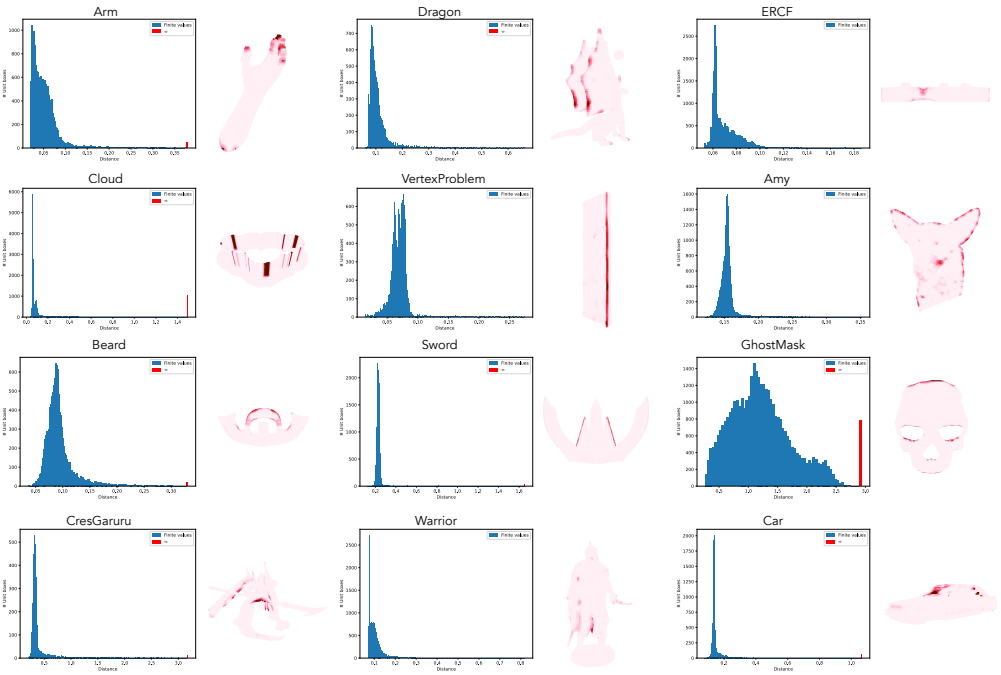


Fig. 10. Output of GlitchFinder on 12 benchmarks with small feature sizes. We omit the axis labels for space — x-axis is distance and y-axis is #unit boxes (same as elsewhere). Dark red regions highlight small features due to which they failed to be sliced correctly. Lines to the right in the distribution graphs indicate that there were unit boxes that had large distances between the two point clouds. Red lines in the distribution graphs indicate that some unit boxes had ∞ distance, meaning that those boxes contained no points in one of the two point clouds. Both of these indicate true differences. The bar for ∞ is chosen such that it is slightly farther on the x-axis from the maximum numeric value distance.

G-code was that it was missing the parts that contained small or intricate features. This suggests that the slicer was able to correct for the bad geometry but failed to handle the small feature size. For example, as reported in the Github issue for the `Arm` model, the user observed that even after repairing the mesh, there was no significant difference in the generated G-code, rendering the use of mesh repair tools on these models ineffective.

This highlights a broader challenge in 3D printing: many imperfections in 3D printing arise due to fine features, yet, existing mesh tools do not typically check for this issue. The reason is that the smallness of a feature is *relative* to the nozzle size and slicing parameters and therefore its effects only manifest *after* slicing (generating G-code). *GlitchFinder can detect this class of problems precisely because it targets G-code*, which captures the interplay between the 3D model, the choice of slicing software, and the settings used to configure the slicer. GlitchFinder is capable of identifying differences arising from this combination of factors, all of which play a crucial role in the actual 3D printing process. This is analogous to static analysis tools that verify bytecode instead of source code, thereby not needing to trust the compiler.

7.3.2 Non-Watertight Models. Figure 11 and Figure 12 show the overall heatmaps and distribution graphs for 33 models which have geometric defects preventing watertight closure. Of the 33 models, 7 (`Airplane`, `Frame`, `Gun`, `Man`, `Tabby`, `Tray`, and `Truss`) were scaled by users to expose

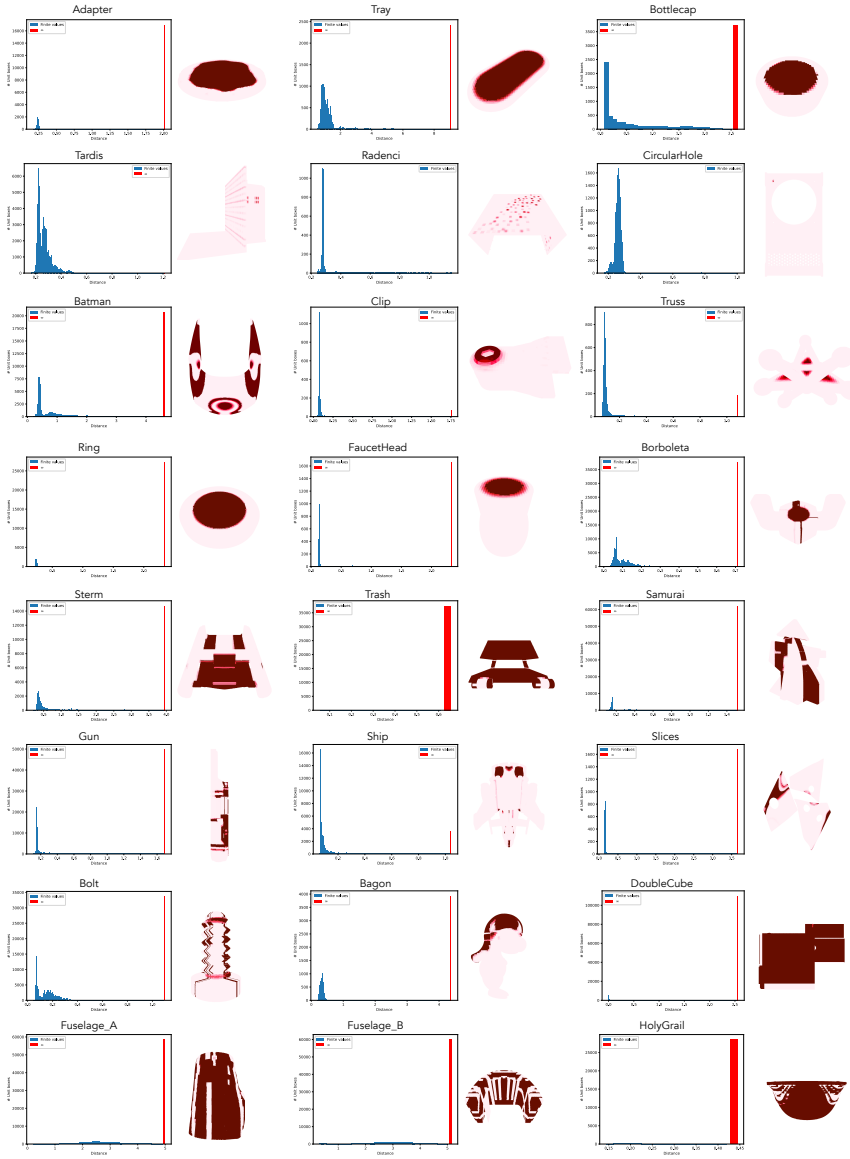


Fig. 11. Output of GlitchFinder on the 24 of 33 non-watertight benchmarks. The dark red regions highlight parts that caused slicing to fail. We omit the axis labels for space — x-axis is distance and y-axis is #unit boxes. Lines to the right in the distribution graphs and red lines have the same meaning as Figure 10.

issues; we replicated those scalings. Non-water-tightness accounts for a significant fraction of slicing failures, often preventing slicers from reliably generating G-code. While some mesh repair tools [6, 7, 110] can correct for non-water-tightness, they do not all check for the same errors, and there is no guarantee that the repaired models will always result in a correct slicing (Section 8.2). GlitchRunner bypasses these limitations by enabling checking $\mathcal{I}$ rather than locating specific flaws: it identifies regions violating $\mathcal{I}$, regardless of the underlying nature of the mesh defect. User

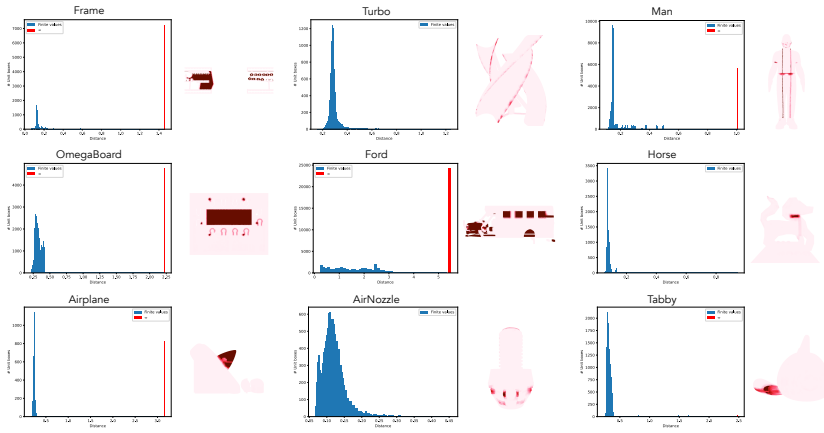


Fig. 12. Continuation of Figure 11: Output of GlitchFinder on the remaining 9 of 33 non-watertight benchmarks. The dark red regions highlight parts that caused slicing to fail. We omit the axis labels for space — x-axis is distance and y-axis is #unit boxes. Lines to the right in the distribution graphs and red lines have the same meaning as Figure 10.

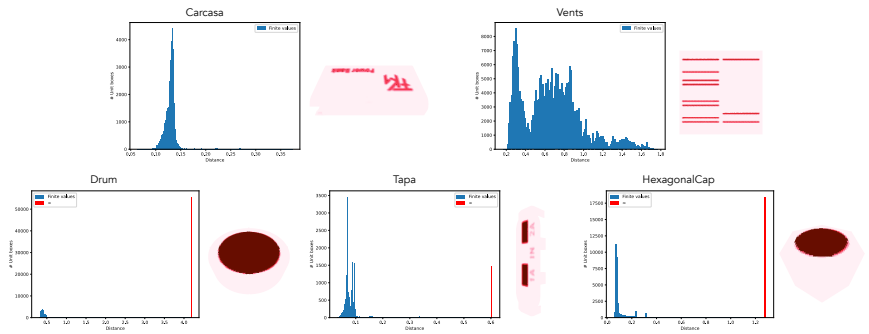


Fig. 13. Output of GlitchFinder on the 5 benchmarks with flipped normals. The dark red regions highlight parts that caused slicing to fail. Lines to the right in the distribution graphs and red lines have the same meaning as Figure 10.

reports also indicate that some models in this category (e.g., `Tardis` and `Ford`) could be successfully repaired by specific mesh repair tools, yielding correct slicing results. However, other models like `Fuselage_A` and `Man` are still incorrectly sliced even after mesh repair.

7.3.3 Models with Flipped Normals. Figure 13 shows the heatmaps and distribution graphs for 5 out of the 50 models for which the only issue was "flipped normals." A normal is a vector perpendicular to the faces of a triangle mesh and typically points outward from the surface, indicating the "outside" of the model. In case of a flipped normal, this perpendicular vector points "inwards" which can confuse slicers into filling a region which is supposed to be a hole. Applying a rotation that orients the hole in a direction that is not orthogonal to the slicing direction can frequently resolve the issue. Therefore, checking $\mathcal{I}$ is well-suited for detecting models with such problems. In Cura, flipped normal issues are not immediately evident upon loading the model unlike non-watertight models which are more visibly flagged by an error message. Instead, users must manually identify these

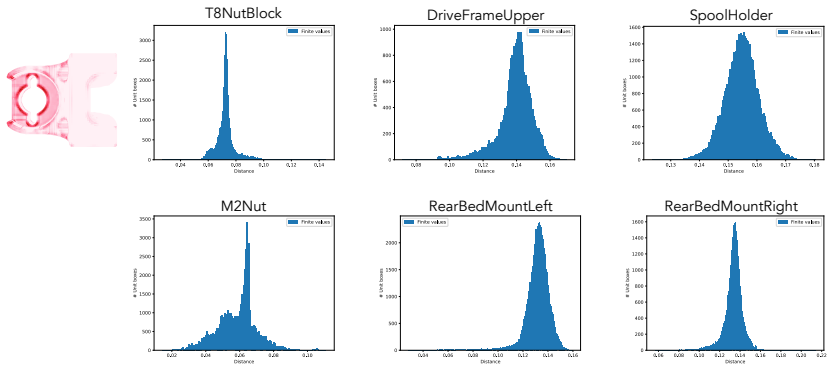


Fig. 14. Output of GlitchFinder on 6 error-free models: the slightly left-skewed / approximate normal distance graph suggests that the differences are due to unwanted errors.

problems by inspecting the preview of the G-code (that most slicers support) to detect "overhangs" that appear in unusual locations, such as on the top surface of the model. This is particularly tedious for large or complex models but something that checking $\mathcal{I}$ with GlitchFinder can easily detect.

7.3.4 Addressing False Positives with Distribution Graphs. By design, GlitchFinder highlights regions with relatively significant deviations which could indicate true differences but also unwanted errors (Section 4.1, Section 5). These unwanted errors can be viewed as being analogous to false positives in traditional program analyses. The heatmaps on their own cannot distinguish between these two cases. To address this, GlitchFinder also generates the distance distribution graph.

We ran GlitchFinder on 6 well-tested, error-free models obtained from a repository that contains parts of a 3D printer [111–116] (such mechanical parts are reasonably error-free) to demonstrate that heatmaps alone are not sufficient for identifying true differences and the distribution graphs are required for effectively distinguishing between true differences and unwanted errors (Figure 14). All distributions appear approximately normal or slightly left-skewed, which is consistent with our earlier observations (Section 5.3) that true differences tend to have an outliers characteristic. For one example (`T8NutBlock`), we show both the heatmap and distribution graph — the heatmap shows that the areas with the darkest colors correspond to the edges or boundaries of the inner hole in the design. These regions show discretization artifacts [32], where the slicer attempts to approximate the circular geometry using discrete linear plastic extrusions.

Summary. The key takeaway from this evaluation is that checking $\mathcal{I}$ with GlitchFinder can localize problematic parts in 3D models arising due to different types of errors. GlitchFinder's heatmap visualization provides an actionable diagnostic tool for the 3D printing workflow, enabling users to address slicing challenges through multiple strategies. For locally problematic regions (e.g., thin features marked in heatmaps), thickening those parts, adjusting parameters (minimum line width, resolution), or changing to a smaller nozzle can help mitigate the problem.

We note that goal for GlitchFinder is not to replace mesh repair tools and other debugging strategies — rather it can serve as a guide to help users identify the root cause of a failed slicing attempt by localizing the error-prone parts of their model. While there is no single tool that can detect all kinds of problems, GlitchFinder is the first tool that covers a much wider range of errors than existing mesh or design level tools. Since GlitchFinder reasons about G-code, it can detect errors that arise due to the effect of slicing parameters and other slicer-level factors.

7.4 Running Times

We ran each benchmark three times, and report the running times as mean ± 95% confidence interval. Across all 56 benchmarks, the running times span 182 ± 2.3s for the fastest model (`Frame`) to 2055 ± 9.2s for the slowest (`Man`). A per-model breakdown of the running times is in Table 8 and Table 9 in Section 11. Since no directly comparable prior tool exists, we compare the running time against the printing time estimates provided by the Cura slicer in Figure 15. For most models, GlitchFinder completes its invariant checks much faster than the actual print, especially for larger ones like `Batman` (see Table 8). Three of the small models `Arm`, `DoubleCube`, `Beard` print too quickly for analysis time to be faster. However, the printing time estimates typically exclude ancillary delays (bed heating, nozzle cooling, etc.) which would only add to the actual print times.

8 Differential Testing of Fabrication Tools

This section evaluates our second claim that by comparing gcode_d^h programs, GlitchFinder finds differences between slicers and can evaluate the efficacy of mesh repair tools. For both cases, we validated the differences highlighted in the heatmaps by manually performing a visual comparison of the G-code and the model.

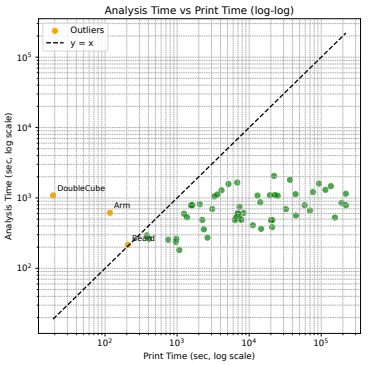


Fig. 15. Comparing GlitchFinder's running time to estimated print times across all 56 benchmarks. Plots are shown in log scale, time is in seconds.

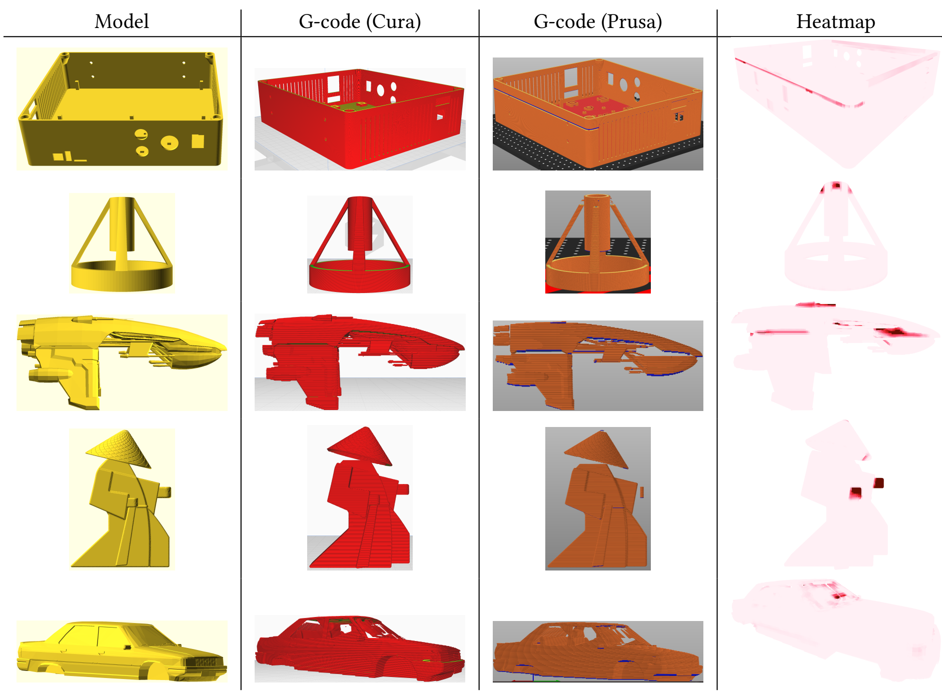
8.1 Comparing Slicers using GlitchFinder

Slicers can produce different G-code for the same model because slicing is fundamentally a path planning problem. They use heuristics to optimize for continuous extrusion, fewer unnecessary movements, and fabrication time. These choices can lead to different outcomes depending on the slicer and model. Some slicers are also tailored for specific printers, limiting their generality. Even with the same slicer, different settings (e.g., seam placement) can change start and end points for layers, resulting in different G-code. We therefore used GlitchFinder to compare the behaviors of two widely used slicers, Cura-5.3.1 and PrusaSlicer-2.7.4, running on MacOS M1, on a set of problematic 3D models. The goal is to identify differences in slicing outcomes and understand how each slicer handles specific geometries.

8.1.1 Benchmarks and setup. We used the 50 problematic models from Section 7, as well as two additional models (`Haut`, `DrillJig`) from PrusaSlicer's issues that can be sliced correctly in Cura but not in PrusaSlicer (52 models in total). We imported each model into both slicers, generated the G-code, and ran GlitchFinder on it. We configured both slicers with the same settings.

8.1.2 Results. For 12 of the 52 models, Cura and PrusaSlicer produced similar G-code based on visual inspection. The remaining 40 models — including all models from Section 7.3.1 and 7.3.3, as well as 21 models from 7.3.2 — demonstrate differences in slicing behaviors between the two slicers. Specifically, PrusaSlicer fails to generate G-code for 2 models (`GhostMask` and `DoubleCube`), resulting in no heatmap for these cases. For 2 models both slicers resulted in incorrect slicing but in different, non-overlapping parts of the model. Cura produces G-code that more accurately represents the original model for 16 of the other 36 models, while PrusaSlicer produces more accurate G-code than Cura on the rest of the 20 (out of 36) models. We show detailed results on 10 representative models, including the CAD design, rendered G-code from Cura and PrusaSlicer and the corresponding heatmaps. For 5 of these Cura-produced G-code more faithfully represents the original model compared to PrusaSlicer, while the other 5 are the opposite.

Table 1. Comparison of slicing results between Cura and PrusaSlicer for five representative models where Cura outperforms PrusaSlicer. The leftmost column shows each model's CAD design in OpenSCAD, followed by G-code visualizations from Cura (middle-left) and PrusaSlicer (middle-right), and heatmaps in the rightmost column. While Cura produces better results overall compared to PrusaSlicer, only the first two models are sliced correctly, and the remaining three models showed slicing errors (Section 7), demonstrating that both slicers struggle with these particular geometries despite Cura's relative advantage.



The first 2 models included in Table 1 are `Haut` and `DrillJig`. Like the 3rd and 4th model in the table, both these models exhibit non-watertight mesh issues. Among the 21 non-watertight models that exhibit differences in slicing behavior, Cura outperforms PrusaSlicer in 14 cases, including a model for which PrusaSlicer fails to generate G-code. PrusaSlicer produces G-code that more precisely represents the model in 9 cases; for example, the last 2 models in Table 2. Although Cura might appear to have an advantage in handling non-watertight models, only 1 out of 5 models with flipped normal issues is handled better in Cura.

The remaining models are all handled better by PrusaSlicer — it resolved the issues and generated G-code, as demonstrated by the first model in Table 2. Among the 12 models with small features, excluding 1 model for which PrusaSlicer fails to generate G-code and 2 models where the outcome is difficult to determine, PrusaSlicer performs better in 7 cases. Two of these cases are highlighted as the 2nd and 3rd models in Table 2. In contrast, Cura generates better slicing results in only 2 cases, one of which is showcased as the final model in Table 1.

Overall, we conclude that Cura and PrusaSlicer are good at handling different kinds of problematic models — Cura was able to handle the non-watertight models better than PrusaSlicer but was outperformed by the latter on the flipped normal models and small-feature models.

8.1.3 Summary. This study shows how GlitchFinder can be used to differentially test slicers across a diverse set of 3D models. The output heatmaps correctly highlighted the discrepancies

Table 2. Comparison of slicing results between Cura and PrusaSlicer for five representative models where PrusaSlicer outperforms Cura. The leftmost column shows each model's geometry in OpenSCAD, followed by G-code visualizations from PrusaSlicer (middle-left) and Cura (middle-right), with heatmaps in the rightmost column. PrusaSlicer achieves completely correct slicing for only the first and last models, while producing more accurate results than Cura for the remaining three cases.

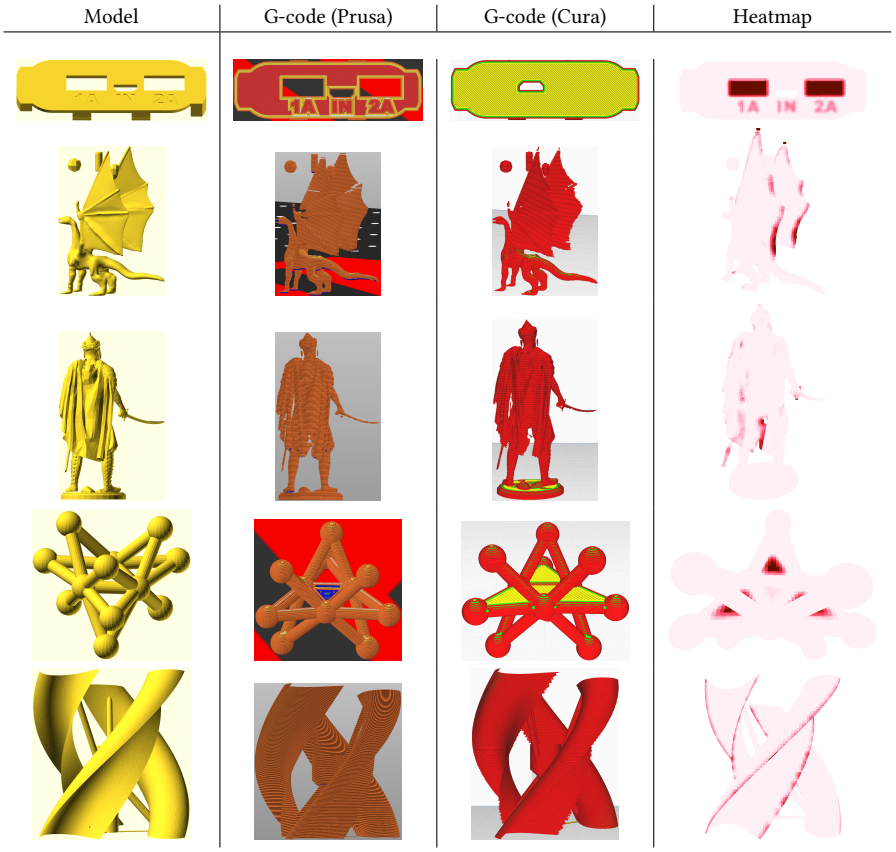
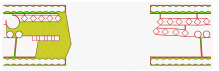
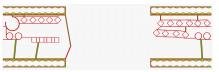


Table 3. Comparing Meshmixer and MeshLab on 37 benchmarks that had no suggested fix on their corresponding Github issue. Note that for 3 models not accounted for in this table, Meshmixer showed a combination of partial improvement with newly introduced defects.

Mesh Tool	Complete resolution	Partial improvement	Not fixed	New slicing defects
MeshLab	5.4% (2/37)	5.4% (2/37)	78.3% (29/37)	10.8% (4/37)
Meshmixer	8.1% (3/37)	8.1% (3/37)	45.9% (17/37)	29.7% (11/37)

between the two programs in all cases. GlitchFinder can be used by developers to compare slicers. For users of slicers, GlitchFinder can act as a guide for deciding which slicer to select for their model. GlitchFinder can empower both developers to improve their software and users to achieve optimal slicing results. Indeed, users can compare G-code produced by the same slicer with different settings to evaluate the effects of configuration settings on slicing quality.

Table 4. Comparison of slicing outcomes before and after MeshLab's repair. Columns show: (1) original CAD model, (2) G-code from defective mesh, (3) G-code from repaired mesh, and (4) heatmap. MeshLab successfully resolved all mesh defects, producing a fully correct slicing result.

Model	G-code (Original)	G-code (MeshLab)	Heatmap
			

8.2 Comparing Mesh Repair Tools using GlitchFinder

It is common in the fabrication community to use mesh repair tools to fix meshes and re-slice when the original model fails to slice. We compare two popular mesh repair tools: MeshLab [110] and Meshmixer2017 [6] to evaluate their effectiveness in fixing problematic 3D models that fail to slice correctly in the Cura slicer.

8.2.1 Benchmarks and setup. We selected 37 models from the 50-model benchmark suite in Section 7 for which a viable repair solution was not already suggested by someone on the corresponding GitHub issue. These models cover all three defect categories (small features, flipped normals, and not-watertight models) in Section 7. For each model, we used both Meshmixer and MeshLab to repair the mesh. We used Meshmixer's "inspector" feature that can auto-repair all errors. MeshLab has many mesh repair functionalities of which we applied 5 fixes to each model: removing duplicate faces, removing duplicate vertices, removing unreferenced vertices, removing zero area faces, and repairing non-manifold edges. We then sliced both repaired meshes with Cura and compare the G-code against the original (uncorrected) G-code using GlitchFinder. We define 4 outcomes:

- **Complete resolution**: All original slicing errors are corrected with no residual defects.
- **Partial improvement**: A subset of the original slicing errors persists.
- **Not fixed**: The repaired mesh has identical slicing failures as the original mesh.
- **New slicing defects**: New errors emerged that were not present for the original mesh.

8.2.2 Results. Table 3 summarizes the results. For both tools, many models remained unfixed for slicing purposes. Notably, for Meshmixer, 8.1% (3 models) showed the combination of partial improvement with newly introduced defects. For both tools, all successfully repaired models (complete or partial fixes) were non-watertight, suggesting that non-watertight models are a primary source of repairable slicing errors. Meshmixer also tends to change geometrically valid models, particularly those containing fine features (e.g., `CresGaruru` and `BearingInsert`). In these cases, Meshmixer not only failed to resolve the original slicing defects, but also either introduced extraneous features or, more often, removed features from the original model, thus ultimately producing worse slicing outcomes. For both tools, we show two representative cases each: one demonstrating successful repair (Table 4, Table 6) and one showing worsened slicing after repair (Table 5, Table 7).

8.2.3 Summary. This showed yet another application of GlitchFinder— by enabling G-code comparison it allowed us to compare the efficacy of popular mesh repair tools when used in the context of slicing models for 3D printing. We found that both mesh repair tools tend to have many "false negatives", i.e., they do not successfully fix the models even when they generate an output.

9 Related Work

Programming languages for fabrication Prior work has used program synthesis, term rewriting, and decompilation techniques to reconstruct CAD models from polygon meshes [4, 16, 18, 29, 52, 72,

Table 5. Comparison of slicing outcomes before and after MeshLab's repair. Columns show: (1) original CAD model, (2) G-code from defective mesh, (3) G-code from repaired mesh, and (4) heatmap. MeshLab introduced critical errors in this case, incorrectly filling all designed holes in the model. The original G-code only exhibited partial slicing defects at a single circular corner (top-left) feature.

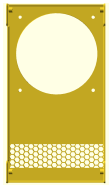
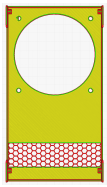
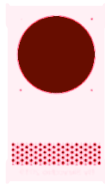
Model	G-code (Original)	G-code (MeshLab)	Heatmap
			

Table 6. Comparison of slicing outcomes before and after Meshmixer's repair. Columns show: (1) original CAD model, (2) G-code from defective mesh, (3) G-code from repaired mesh, and (4) heatmap. Meshmixer successfully addressed all geometric flaws, resulting in a correct slicing.

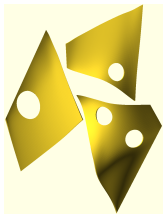
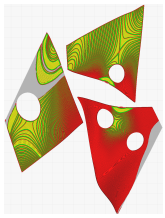
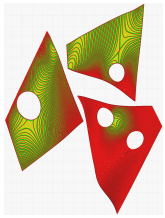
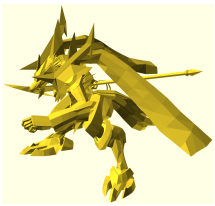
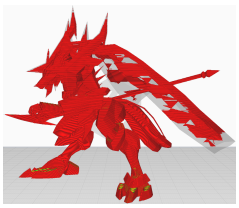
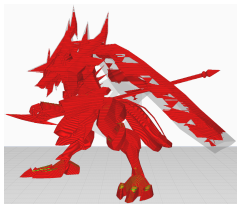
Model	G-code (Original)	G-code (Meshmixer)	Heatmap
			

Table 7. While manually corrected for manifold errors by the user, this model still showed slicing failures in fine features (e.g., thin cape). Meshmixer further degraded results by removing certain mesh elements at the model's foot and ankle area, which can be clearly seen in the heatmap.

Model	G-code (Original)	G-code (Meshmixer)	Heatmap
			

74, 86, 99], provided a formal semantics for CSG and mesh [73, 87] which has been used to develop semantics preserving compilers and decompilers [73], and laid the foundation for developing tools like debuggers and analyzers for toolpaths [103]. Vespidae [36] allows users to develop and visualize custom toolpaths. Imprimer [101] has explored literate programming in the context of CNC milling. In concurrent work, Tekriwal and Sottile [98] presented the first mechanized semantics of a subset of G-code and a formally verified interpreter in Why3 and Rocq. Our definition of state, σ, is similar to the machine's physical state defined by Tekriwal and Sottile [98] although they do not "lift" the

G-code commands by modeling them as cuboids. However, none of these formally reason about G-code for checking invariants of models or differential testing of fabrication tools.

Understanding G-code. GSim (Tsoutsos et al. [104]) decompiles G-code to obtain an approximation of the input as a CSG model. We implemented Tsoutsos et al. [104]'s approach and found that their method for reconstructing G-code using CSG does not scale for comparing G-code programs (Section 3.2). Recent work [8, 50, 64] has used large language models (LLMs) for debugging and comprehension of G-code and for CAD/CAM more broadly. García Domínguez et al. [38] developed an algorithm for constructing point clouds from scanned objects and compiling them directly to toolpaths. We start with G-code (which is closer to toolpaths) and reconstruct a point cloud modeling the lines in the G-code. Yanamandra et al. [122] used machine learning to reverse engineer the toolpath used to manufacture a 3D object using fiber reinforced ABS filament. Baumann et al. [9] developed a simulation technique based on contour detection for reverse engineering STL meshes from G-code. Prior work has studied techniques for optimizing the orientation of a model when 3D printing in order to increase mechanical strength [26, 106], and developed methods for analyzing the quality of 3D printed objects [118]. While in this work we targeted uniform, planar slicing [28, 60], other slicing techniques [32, 39] are witnessing adoption in many tools [59].

Studying slicers. Šljivic et al. [124] and Mohd Ariffin et al. [69] compared slicers by analyzing printed objects, while Bryła [12] examined how filament, slicer, and printer together affect output. These works studied the manufactured results rather than the G-code itself. Bryła and Martowicz [13] conducted a *syntactic* comparison of G-code from two slicers [23, 81], examining how identical settings yield different artifacts. In contrast, we use GlitchFinder to *semantically* analyze slicer behavior, highlighting differences in how each handles geometric challenges. Recent work [95] explores interactive tuning via parameter adjustment.

10 Conclusion

In this work, we lay the groundwork for formal reasoning of toolpaths in the form on G-code. We defined semantics for linear-motion G-code and developed GlitchFinder, a static analysis tool for comparing G-code programs. We showed two key applications of GlitchFinder: checking model invariants and differential testing of slicers and mesh repair tools and demonstrated both on real-world models and popular tools. By operating directly on G-code, GlitchFinder enables rigorous analysis of slicer behavior and can be integrated into CI pipelines to monitor tooling quality. Future directions include supporting additional slicing strategies and exploring further applications of high-level geometry reconstruction from G-code.

Acknowledgments

Thanks to the anonymous reviewers for their feedback. We are grateful to Zachary Tatlock for sharing valuable, insightful comments about the work and to Amy Zhu for reading an earlier draft.

Data Availability Statement

The code and data used in this paper are available on Zenodo [42]. The code is developed on GitHub [43, 44]. The benchmarks we used are all public, though we do not own them [45]. Table 8, Table 9 in the submitted supplementary material (Section 11) have links to all benchmarks and they are included as part of the artifact.

11 Parameters for running GlitchFinder

The parameters used in our evaluation are shown in Table 8 and Table 9. The value of Sampling Gap depends on the size of the smallest feature in the model. Selecting an appropriate Unit Box Size is crucial, as it helps isolate errors and ensures the heatmaps accurately highlight problematic regions.

The right value will obtain the "right" number of errors – a number which does not overwhelm the user but still provides "actionable" information, a key criteria for success in traditional static analysis tools. We list the values that we found worked best for each model.

Table 8. Parameters used for all 58 benchmarks in our evaluation: 50 models with errors, 6 error-free Voron-0 models, and 2 additional models we used for comparing Prusa and Cura slicers (Section 8.1). All dimensions, sampling gaps, and unit box sizes are given in millimeters (mm). Running time of GlitchFinder for the invariant checking application (Section 7) in seconds is shown in the last column. `DrillJig` and `Haut` were not included in the invariant checking benchmark suite, so no runtime data are reported for them. Times are presented as mean ± 95% confidence interval, rounding the mean up to the next whole second and the interval up to the next-highest tenth of a second.

Name	Dimensions	Sampling Gap	Unit Box Size	Time (s)
Adapter [84]	130.0 × 130.0 × 6.7	0.5	1.0 × 1.0 × 1.0	485 ± 8.1
AirNozzle	19.4 × 19.3 × 31.3	0.08	0.6 × 0.6 × 0.6	530 ± 1.6
Airplane [62]	98.8 × 41.1 × 65.9	0.5	1.5 × 1.5 × 1.5	483 ± 2.0
Amy [27]	76.9 × 88.3 × 5.0	0.3	1.0 × 1.0 × 0.5	607 ± 3.1
Arm [78]	11.8 × 8.9 × 10.2	0.03	0.2 × 0.2 × 0.2	616 ± 0.2
Bagon [107]	60.8 × 83.2 × 112.9	1.0	2.0 × 2.0 × 2.0	564 ± 1.0
Batman [15]	189.9 × 239.6 × 155.0	1.5	2.0 × 2.0 × 2.0	1600 ± 9.7
Beard [49]	27.5 × 10.5 × 8.1	0.05	0.3 × 0.3 × 0.3	215 ± 1.8
BearingInsert [33]	40.0 × 25.0 × 7.0	0.08	0.6 × 0.6 × 0.6	785 ± 6.0
Bolt [53]	28.0 × 27.9 × 43.0	0.1	0.3 × 0.3 × 0.8	1119 ± 10.0
Borboleta [51]	20.8 × 38.0 × 20.0	0.1	0.3 × 0.3 × 0.3	813 ± 7.0
Bottlecap [34]	33.0 × 33.0 × 10.3	0.1	1.0 × 1.0 × 1.0	358 ± 0.5
Carcasa [35]	100.0 × 61.0 × 22.4	0.2	0.8 × 0.8 × 0.5	1081 ± 3.1
Car [94]	36.4 × 78.7 × 21.8	0.2	0.8 × 0.8 × 0.8	483 ± 4.4
CircularHole [21]	110.0 × 200.0 × 12.0	0.5	1.0 × 1.0 × 1.0	409 ± 3.6
Clip [93]	28.8 × 40.1 × 21.9	0.1	1.0 × 1.0 × 1.0	697 ± 3.8
Cloud [19]	41.5 × 66.8 × 15.0	0.08	0.3 × 0.3 × 1.0	791 ± 1.0
CresGaruru [10]	138.6 × 136.8 × 98.1	1.0	2.0 × 2.0 × 2.0	364 ± 1.1
DoubleCube [17]	3.6 × 3.3 × 2.8	0.01	0.05 × 0.05 × 0.05	1087 ± 2.4
Dragon [123]	14.5 × 29.5 × 36.3	0.1	0.5 × 0.5 × 0.5	237 ± 3.1
DrillJig [67]	57.0 × 57.1 × 48.3	0.1	0.8 × 0.8 × 0.8	N/A
DriveFrameUpper [111]	61.0 × 50.0 × 22.5	0.2	1.0 × 1.0 × 1.0	510 ± 2.6
Drum [31]	163.0 × 163.0 × 38.0	1.2	2.0 × 2.0 × 2.0	1148 ± 7.0
Fuselage_A [56]	99.5 × 100.5 × 133.4	1.5	2.0 × 2.0 × 2.0	789 ± 6.9
Fuselage_B [70]	120.3 × 137.2 × 85.8	1.0	2.0 × 2.0 × 2.0	851 ± 4.5
FaucetHead [14]	35.0 × 34.9 × 42.0	0.2	1.0 × 1.0 × 1.0	521 ± 2.5
Ford [68]	244.7 × 153.4 × 113.1	0.5	2.5 × 2.5 × 2.5	1213 ± 3.0
Frame [40]	260.9 × 75.0 × 0.6	0.1	1.0 × 1.0 × 0.3	182 ± 2.3
GhostMask [91]	150.0 × 70.6 × 162.3	0.6	1.5 × 1.5 × 1.5	868 ± 2.2
Gun [80]	32.0 × 27.8 × 212.1	0.3	0.6 × 0.6 × 1.0	1092 ± 16.6
Haut [48]	247.3 × 67.5 × 211.0	0.85	3.0 × 3.0 × 1.0	N/A
HexagonalCap [82]	20.9 × 24.1 × 17.2	0.1	0.5 × 0.5 × 0.5	487 ± 3.9
HolyGrail [3]	15.2 × 15.2 × 6.3	0.05	0.3 × 0.3 × 0.3	294 ± 1.7
Horse [37]	43.5 × 49.2 × 53.6	0.1	0.8 × 0.8 × 0.8	1656 ± 25.5
M2Nut [112]	145.0 × 5.5 × 2.4	0.1	1.0 × 0.2 × 0.2	255 ± 1.5
Man [22]	58.8 × 32.0 × 149.6	0.3	0.6 × 0.6 × 0.6	2055 ± 9.2
OmegaBoard [58]	245.0 × 191.0 × 13.2	0.8	1.2 × 1.2 × 1.0	693 ± 4.0
Ring [120]	90.0 × 90.0 × 10.0	0.5	1.0 × 1.0 × 1.0	384 ± 3.1
Radenci [25]	250.3 × 250.0 × 150.0	0.5	3.0 × 3.0 × 3.0	789 ± 7.3

Table 9. Continuation of Table 8: Parameters used for all 58 benchmarks in our evaluation: 50 models with errors, 6 error-free Voron-0 models, and 2 additional models we used for comparing Prusa and Cura slicers. All dimensions, sampling gaps, and unit box sizes are given in millimeters (mm). Running time of GlitchFinder in seconds is shown in the last column. Times are presented as mean ± 95% confidence interval, rounding the mean up to the next whole second and the interval up to the next-highest tenth of a second.

Name	Dimensions	Sampling Gap	Unit Box Size	Time (s)
RearBedMountLeft [113]	45.9 × 35.7 × 29.8	0.2	0.6 × 0.6 × 0.6	594 ± 2.6
RearBedMountRight [114]	45.9 × 56.9 × 29.8	0.2	0.8 × 0.8 × 0.8	613 ± 4.2
Samurai [61]	39.4 × 63.5 × 87.3	0.3	0.8 × 0.8 × 0.8	1100 ± 9.7
Ship [2]	36.6 × 27.7 × 65.5	0.1	0.5 × 0.5 × 0.5	1578 ± 27.5
Slices [108]	148.0 × 198.4 × 40.3	0.3	2.0 × 2.0 × 0.5	751 ± 2.1
SpoolHolder [115]	92.0 × 60.0 × 15.0	0.3	1.0 × 1.0 × 0.5	488 ± 3.6
Sterm [20]	115.9 × 147.1 × 50.7	1.0	2.0 × 2.0 × 2.0	662 ± 1.2
Sword [24]	198.1 × 120.6 × 35.7	0.5	1.0 × 1.0 × 1.0	1077 ± 8.9
T8NutBlock [116]	39.2 × 28.0 × 25.0	0.1	0.5 × 0.5 × 0.5	1287 ± 7.6
Tabby [63]	96.8 × 140.7 × 98.7	0.8	2.0 × 2.0 × 2.0	1471 ± 8.8
Tapa [119]	10.7 × 61.0 × 19.5	0.1	0.5 × 0.5 × 0.5	597 ± 4.2
Tardis [85]	100.0 × 129.8 × 110.0	0.4	1.0 × 1.0 × 1.0	1810 ± 6.5
Trash [75]	15.9 × 17.2 × 9.8	0.05	0.3 × 0.3 × 0.3	264 ± 2.2
Tray [109]	295.8 × 295.8 × 10.8	3.0	5.0 × 5.0 × 1.0	527 ± 2.2
Truss [66]	24.7 × 21.8 × 18.5	0.1	0.5 × 0.5 × 0.5	263 ± 0.2
Turbo [97]	150.0 × 150.0 × 150.0	0.8	2.0 × 2.0 × 2.0	1130 ± 6.2
Vents [100]	140.0 × 140.0 × 22.0	0.8	1.2 × 1.2 × 1.2	1306 ± 7.3
VertexProblem [105]	17.2 × 4.0 × 60.0	0.1	0.5 × 0.5 × 1.0	273 ± 3.0
Warrior [121]	35.7 × 34.1 × 57.2	0.1	0.7 × 0.7 × 0.7	1052 ± 4.1

References

[1] Rep-rap Contributors . 2025. G-code - RepRap. https://reprap.org/wiki/G-code

[2] Aceface. 2021. Model after slicing has gaps. Any tips how to fix? https://forum.prusa3d.com/forum/prusaslicer/model-after-slicing-has-gaps-any-tips-how-to-fix/.

[3] AircreteHarry. 2024. Cura makes rounded parts flat after slicing and some parts disappear. https://community.ultimaker.com/topic/45609-cura-makes-rounded-parts-flat-after-slicing-and-some-parts-disappear-anyone-know-what-setting-to-use-to-fix-problem/#comment-334277.

[4] John Altidor, Jack Wileden, Jeffrey McPherson, Ian Grosse, Sundar Krishnamurty, Felicia Cordeiro, and Audrey Lee-St. John. 2011. A Programming Language Approach to Parametric CAD Data Exchange *(IDETCCIEC, Vol. Volume 2: 31st Computers and Information in Engineering Conference)*. 779–791. https://doi.org/10.1115/DETC2011-48530

[5] N. Aspert, D. Santa-Cruz, and T. Ebrahimi. 2002. MESH: measuring errors between surfaces using the Hausdorff distance. In *Proceedings. IEEE International Conference on Multimedia and Expo*, Vol. 1. 705–708 vol.1. https://doi.org/10.1109/ICME.2002.1035879

[6] Autodesk App Store. 2017. MakePrintable Mesh Repair for Fusion 360. https://apps.autodesk.com/FUSION/en/Detail/Index?id=4108920185261935100&appLang=en&os=Win64. Accessed: 2025-03-25.

[7] Autodesk Inc. 2025. Autodesk Netfabb. https://www.autodesk.com/products/netfabb/overview. Accessed: 2025-03-25.

[8] Silvia Badini, Stefano Regondi, Emanuele Frontoni, and Raffaele Pugliese. 2023. Assessing the capabilities of ChatGPT to improve additive manufacturing troubleshooting. *Advanced Industrial and Engineering Polymer Research* 6, 3 (2023), 278–287. https://doi.org/10.1016/j.aiepr.2023.03.003

[9] Felix W Baumann, Martin Schuermann, Ulrich Odefey, and Markus Pfeil. 2017. From GCode to STL: Reconstruct Models from 3D Printing as a Service. *IOP Conference Series: Materials Science and Engineering* 280, 1 (dec 2017), 012033. https://doi.org/10.1088/1757-899X/280/1/012033

[10] Beelzemon. 2024. Pieces of model disappear upon slicing. https://community.ultimaker.com/topic/44843-pieces-of-model-disappear-upon-slicing/#comment-329547.

[11] T. Birsan and D. Tiba. 2006. One Hundred Years Since the Introduction of the Set Distance by Dimitrie Pompeiu. In *System Modeling and Optimization*, F. Ceragioli, A. Dontchev, H. Futura, K. Marti, and L. Pandolfi (Eds.). Springer US, Boston, MA, 35–39.

[12] Jakub Bryła. 2021. The influence of the MEX manufacturing parameters on the tensile elastic response of printed elements. *Rapid Prototyping Journal* 27, 1 (2021), 187–196. https://doi.org/10.1108/RPJ-02-2020-0034

[13] Jakub Bryła and Adam Martowicz. 2021. Study on the Importance of a Slicer Selection for the 3D Printing Process Parameters via the Investigation of G-Code Readings. *Machines* 9, 8 (2021). https://doi.org/10.3390/machines9080163

[14] buteomont. 2024. Model looks different when viewed in Preview vs Prepare. https://github.com/Ultimaker/Cura/issues/19223.

[15] Canon182. 2022. Slicing Incorrectly. https://github.com/Ultimaker/Cura/issues/11733.

[16] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-unification. *Proc. ACM Program. Lang.* 7, POPL, Article 14 (Jan. 2023), 29 pages. https://doi.org/10.1145/3571207

[17] caretdashcaret. 2014. MeshRepairTestModels. https://github.com/caretdashcaret/MeshRepairTestModels/tree/master?tab=readme-ov-file.

[18] Molly Aubrey Carton, Chandrakana Nandi, Adam Anderson, Haisen Zhao, Eva Darulova, Dan Grossman, Jeffrey Ian Lipton, Adriana Schulz, and Zachary Tatlock. 2021. A Roadmap Towards Parallel Printing for Desktop 3D Printers. In *Proceedings of the 2021 International Solid Freeform Fabrication Symposium*. University of Texas at Austin. https://doi.org/10.26153/tsw/17639 Mechanical Engineering Department.

[19] catherinesays. 2023. Slicing problems - Parts of Print Missing. https://forum.prusa3d.com/forum/original-prusa-i3-mmu2s-mmu2-how-do-i-print-this-printing-help/slicing-problems-parts-of-print-missing/.

[20] CRigio83. 2024. Ultimaker doesn't slice all the model. https://community.ultimaker.com/topic/45504-ultimaker-dont-slice-all-the-model/#comment-333597.

[21] crysxd. 2024. Circle geometry damaged in slicing process. https://github.com/Ultimaker/Cura/issues/6998.

[22] CubeBox391. 2022. Some parts of model disappear after slicing. https://community.ultimaker.com/topic/41818-some-parts-of-model-disapear-after-slicing/#comment-313826.

[23] Cura. 2024. Cura Software. https://ultimaker.com/en/products/cura-software.

[24] D0GG0. 2023. Missing parts. https://community.ultimaker.com/topic/44400-missing-parts/.

[25] Darkglowing. 2022. I need help with slicing Archicad model. https://community.ultimaker.com/topic/39792-i-need-help-with-slicing-archicad-model/.

[26] P. Delfs, M. Tows, and H.-J. Schmid. 2016. Optimized build orientation of additive manufactured parts for improved surface quality and build time. *Additive Manufacturing* 12 (2016), 314–320. https://doi.org/10.1016/j.addma.2016.06.003 Special Issue on Modeling & Simulation for Additive Manufacturing.

[27] Delphi-FPC-Lazarus. 2022. Slicing failed with an unexpected error. https://github.com/Ultimaker/Cura/issues/12975.
[28] A. Dolenc and I. Mäkelä. 1994. Slicing procedures for layered manufacturing techniques. *Computer-Aided Design* 26, 2 (1994), 119–126. https://doi.org/10.1016/0010-4485(94)90032-9
[29] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. InverseCSG: automatic conversion of 3D models to CSG trees. *ACM Trans. Graph.* 37, 6, Article 213 (Dec. 2018), 16 pages. https://doi.org/10.1145/3272127.3275006
[30] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19
[31] etherbrat. 2024. Infill leaks out of a manifold solid. https://github.com/Ultimaker/Cura/issues/18079.
[32] Jimmy Etienne, Nicolas Ray, Daniele Panozzo, Samuel Hornus, Charlie C.L. Wang, Jonàs Martínez, Sara Mcmains, Marc Alexa, Brian Wyvill, and Sylvain Lefebvre. 2019. CurviSlicer: Slightly curved slicing for 3-axis printers. *ACM Transactions on Graphics* 38, 4 (Aug. 2019), 1–11. https://doi.org/10.1145/3306346.3323022
[33] Ette. 2024. EnragedRabbitProject. https://github.com/EtteGit/EnragedRabbitProject.
[34] Fabio3rs. 2024. Thread from a bottlecap slicing incorrectly. https://github.com/Ultimaker/Cura/issues/16886.
[35] Faklam. 2019. Power Bank - 18650x3. https://www.thingiverse.com/thing:3661158.
[36] Frikk H Fossdal, Vinh Nguyen, Rogardt Heldal, Corie L. Cobb, and Nadya Peek. 2023. Vespidae: A Programming Framework for Developing Digital Fabrication Workflows. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference* (, Pittsburgh, PA, USA,) *(DIS '23)*. Association for Computing Machinery, New York, NY, USA, 2034–2049. https://doi.org/10.1145/3563657.3596106
[37] Andrew Galbreath. 2023. Missing Layers after slicing. https://forum.prusa3d.com/forum/prusaslicer/missing-layers-after-slicing/.
[38] Amabel García Domínguez, Juan Claver Gil, Jorge Ayllon Perez, Marta Marin, and Eva Rubio. 2023. NC Toolpath Generation and Optimization from 3D Point Cloud in Reverse Engineering. 204–212. https://doi.org/10.4028/p-SRc0bO
[39] Neil Gershenfeld. 2024. Rapid-Prototyping of Rapid-Prototyping Machines:How to Make Something that Makes (almost) Anything. https://fab.cba.mit.edu/classes/865.21/topics/path_planning/additive.html.
[40] Gianpy86. 2021. Slicing error. https://github.com/prusa3d/PrusaSlicer/issues/7620.
[41] Iddo Hanniel, Adarsh Krishnamurthy, and Sara McMains. 2012. Computing the Hausdorff distance between NURBS surfaces using numerical iteration on the GPU. *Graphical Models* 74, 4 (2012), 255–264. https://doi.org/10.1016/j.gmod.2012.05.002 GMP2012.
[42] Yumeng He, Chandrakana Nandi, and Sreepathi Pai. 2025. Artifact for Formalizing Linear Motion G-code for Invariant Checking and Differential Testing of Fabrication Tools. https://doi.org/10.5281/zenodo.16595028
[43] Yumeng He, Chandrakana Nandi, and Sreepathi Pai. 2025. GlitchFinder. GitHub repository. https://github.com/ymh1003/GlitchFinder
[44] Yumeng He, Chandrakana Nandi, and Sreepathi Pai. 2025. GLITCH_scripts. GitHub repository. https://github.com/ymh1003/GLITCH_scripts
[45] Yumeng He, Chandrakana Nandi, and Sreepathi Pai. 2025. tricky-3d-models: A benchmark suite for 3D CAD models that are hard to print. GitHub repository. https://github.com/chandrakananandi/tricky-3d-models Commit history accessed 27 commits as of August 18, 2025.
[46] Nathaniel Hudson, Celena Alcock, and Parmit K. Chilana. 2016. Understanding Newcomers to 3D Printing: Motivations, Workflows, and Barriers of Casual Makers. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 384–396. https://doi.org/10.1145/2858036.2858266
[47] D. P. Huttenlocher, G. A. Klanderman, and W. A. Rucklidge. 1993. Comparing Images Using the Hausdorff Distance. *IEEE Trans. Pattern Anal. Mach. Intell.* 15, 9 (sep 1993), 850–863. https://doi.org/10.1109/34.232073
[48] Jaggana69. 2024. Missing layer. https://github.com/prusa3d/PrusaSlicer/issues/12074.
[49] Jasvijay. 2023. Error Rendering object. https://community.ultimaker.com/topic/44310-error-rendering-object/#comment-326324.
[50] Anushrut Jignasu, Kelly Marshall, Baskar Ganapathysubramanian, Aditya Balu, Chinmay Hegde, and Adarsh Krishnamurthy. 2023. Towards Foundational AI Models for Additive Manufacturing: Language Models for G-Code Debugging, Manipulation, and Comprehension. arXiv:2309.02465 [cs.SE]
[51] jmgk77. 2022. Incorrect slicing. https://github.com/Ultimaker/Cura/issues/11702.
[52] R. Kenny Jones, David Charatan, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. 2021. ShapeMOD: macro operation discovery for 3D shape programs. *ACM Transactions on Graphics* 40, 4 (July 2021), 1–16. https://doi.org/10.1145/3450626.3459821
[53] jvos1. 2024. Sliced model missing layers. https://github.com/Ultimaker/Cura/issues/7316.

[54] Community Klipper. 2024. Klipper 3D. https://www.klipper3d.org/

[55] Thomas Kramer, Frederick Proctor, and Elena Messina. 2000. The NIST RS274NGC Interpreter - Version 3. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=823374

[56] Kryszyn. 2022. Unknown slicing error. https://github.com/Ultimaker/Cura/issues/13339.

[57] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *SIGPLAN Not.* 49, 6 (June 2014), 216–226. https://doi.org/10.1145/2666356.2594334

[58] leonirz. 2024. Wrong PREVIEW. https://github.com/Ultimaker/Cura/issues/5927.

[59] Sungwoo Lim, Richard A. Buswell, Philip J. Valentine, Daniel Piker, Simon A. Austin, and Xavier De Kestelier. 2016. Modelling curved-layered printing paths for fabricating large-scale construction components. *Additive Manufacturing* 12 (2016), 216–230. https://doi.org/10.1016/j.addma.2016.06.004 Special Issue on Modeling & Simulation for Additive Manufacturing.

[60] Marco Livesu, Stefano Ellero, Jonàs Martínez, Sylvain Lefebvre, and Marco Attene. 2017. From 3D models to 3D prints: an overview of the processing pipeline. *Computer Graphics Forum* 36, 2 (2017), 537–564. https://doi.org/10.1111/cgf.13147 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13147

[61] lyrana. 2021. Missing middle section when slicing part on its side? https://community.ultimaker.com/topic/37996-missing-middle-section-when-slicing-part-on-its-side/.

[62] Macattack124. 2022. Missing Part of Model when Sliced in Cura. https://community.ultimaker.com/topic/42198-missing-part-of-model-when-sliced-in-cura/#comment-315758.

[63] Maccraft123. 2023. Missing part of a model after slicing. https://github.com/prusa3d/PrusaSlicer/issues/11124.

[64] Liane Makatura, Michael Foshey, Bohan Wang, Felix HähnLein, Pingchuan Ma, Bolei Deng, Megan Tjandrasuwita, Andrew Spielberg, Crystal Elaine Owens, Peter Yichen Chen, Allan Zhao, Amy Zhu, Wil J Norton, Edward Gu, Joshua Jacob, Yifei Li, Adriana Schulz, and Wojciech Matusik. 2023. How Can Large Language Models Help Humans in Design and Manufacturing? arXiv:2307.14377 [cs.CL]

[65] MarlinFirmware. 2024. Marlin Firmware - A Really Good 3D Printer Driver. https://marlinfw.org/

[66] Matto. 2024. Ultimaker Cura slicing bug. https://forum.core-electronics.com.au/t/ultimaker-cura-slicing-bug/6864/2.

[67] mcboeing. 2019. Prusa Slicer errors. https://github.com/prusa3d/PrusaSlicer/issues/3036.

[68] michael Okeefe91. 2022. Geometry problem with my model. https://forums.sketchup.com/t/geometry-problem-with-my-model/202390.

[69] M K A Mohd Ariffin, N A Sukindar, B.T. H T Baharudin, C N A Jaafar, and M I S Ismail. 2018. Slicer Method Comparison Using Open-source 3D Printer. *IOP Conference Series: Earth and Environmental Science* 114, 1 (jan 2018), 012018. https://doi.org/10.1088/1755-1315/114/1/012018

[70] muellerbru. 2020. Thin Wall not sliced. https://github.com/prusa3d/PrusaSlicer/issues/3708.

[71] James R Munkers. 2000. Topology.

[72] Chandrakana Nandi, Anat Caspi, Dan Grossman, and Zachary Tatlock. 2017. Programming Language Tools and Techniques for 3D Printing. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.10

[73] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional Programming for Compiling and Decompiling Computer-aided Design. *Proc. ACM Program. Lang.* 2, ICFP, Article 99, 31 pages. https://doi.org/10.1145/3236794

[74] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

[75] nikita090309. 2024. Cura deletes my walls. https://github.com/Ultimaker/Cura/issues/18934.

[76] OnShape. 2025. The Leader in Cloud-Native CAD & PDM. https://www.onshape.com/en//.

[77] OpenScad. 2014. OpenScad. The Programmers Solid 3D CAD Modeller. http://www.openscad.org/.

[78] PolygonBronson. 2022. Missing Layers in Preview (v5.1.0) when Previous Version (v4.13.1) Showed with Same Slicer Settings. https://github.com/Ultimaker/Cura/issues/12967.

[79] Alina Pranovich, Sasan Gooran, Jeppe Revall Frisvad, and Daniel Nyström. 2020. Surface discretisation effects on 3D printed surface appearance. *CEUR Workshop Proceedings* 2688. https://doi.org/10.5281/zenodo.4081343 10th Colour and Visual Computing Symposium , CVCS 2020 ; Conference date: 16-09-2020 Through 17-09-2020.

[80] Raven_CDN. 2021. Missing sections of a print in Preview. https://community.ultimaker.com/topic/36060-missing-sections-of-a-print-in-preview/.

[81] Research Prusa. 2024. An open-source, feature-rich, frequently updated tool that contains everything you need to export the perfect print files for your 3D printer. https://www.prusa3d.com/en/page/prusaslicer_424/.

[82] rftghost. 2019. Incorrect slicing of threaded object. https://github.com/Ultimaker/Cura/issues/5582.

[83] Rhinoceros. 2024. Design, model, present, analyze, realize. https://www.rhino3d.com/.

[84] RobertBagy. 2024. Slicing adds extra roof. https://github.com/Ultimaker/Cura/issues/18913.

[85] sadovsf. 2021. Slicer creates holes in sliced model. https://github.com/prusa3d/PrusaSlicer/issues/6290.

[86] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. 2017. CSGNet: Neural Shape Parser for Constructive Solid Geometry. *CoRR* abs/1712.08290 (2017). arXiv:1712.08290 http://arxiv.org/abs/1712.08290

[87] Benjamin Sherman, Jesse Michel, and Michael Carbin. 2019. Sound and robust solid modeling via exact real arithmetic and continuity. *Proc. ACM Program. Lang.* 3, ICFP, Article 99 (July 2019), 29 pages. https://doi.org/10.1145/3341703

[88] Simplify3D. 2024. Unleash the Power of Additive Manufacturing. https://www.simplify3d.com/.

[89] Slic3r. 2024. Slic3r: Open source 3D printing toolbox. http://slic3r.org/.

[90] P. Smid. 2003. *CNC Programming Handbook: A Comprehensive Guide to Practical CNC Programming*. Industrial Press. https://books.google.com/books?id=JNnQ8r5merMC

[91] Sniper-1818. 2023. Unable to slice (tried 5.4 and 5.5). https://github.com/Ultimaker/Cura/issues/17216.

[92] Solidworks. 2024. Smarter. Faster. Together. http://www.solidworks.com/.

[93] stemsin. 2024. Part of the model is visible in the prepare mode, but does not come in preview. https://github.com/Ultimaker/Cura/issues/13831.

[94] suatbatu. 2023. Car slice failure. https://github.com/Ultimaker/Cura/issues/17571.

[95] Blair Subbaraman, Nathaneal Bursch, and Nadya Peek. 2025. It's Not the Shape, It's the Settings: Tools for Exploring, Documenting, and Sharing Physical Fabrication Parameters in 3D Printing. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '25)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3706598.3713354

[96] Pavel Surynek, Vojtěch Bubník, Lukáš Matěna, and Petr Kubiš. 2025. Object Packing and Scheduling for Sequential 3D Printing: a Linear Arithmetic Model and a CEGAR-inspired Optimal Solver. arXiv:2503.05071 [cs.CG] https://arxiv.org/abs/2503.05071

[97] Ted5855. 2023. Slicing. https://github.com/Ultimaker/Cura/issues/15653.

[98] Mohit Tekriwal and Matthew Sottile. 2025. Mechanized Semantics for Correctness of the RS274 Additive Manufacturing Command Language. In *NASA Formal Methods*, Aaron Dutle, Laura Humphrey, and Laura Titolo (Eds.). Springer Nature Switzerland, Cham, 341–359. https://doi.org/10.1007/978-3-031-93706-4_20

[99] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Infer and Execute 3D Shape Programs. In *International Conference on Learning Representations*.

[100] tispokes. 2024. Slicing incorrect – closes some holes randomly. https://github.com/Ultimaker/Cura/issues/17797.

[101] Jasper Tran O'Leary, Gabrielle Benabdallah, and Nadya Peek. 2023. Imprimer: Computational Notebooks for CNC Milling. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 207, 15 pages. https://doi.org/10.1145/3544548.3581334

[102] Jasper Tran O'Leary, Eunice Jun, and Nadya Peek. 2022. Improving Programming for Exploratory Digital Fabrication with Inline Machine Control and Styled Toolpath Visualizations. In *Proceedings of the 7th Annual ACM Symposium on Computational Fabrication* (Seattle, WA, USA) *(SCF '22)*. Association for Computing Machinery, New York, NY, USA, Article 8, 12 pages. https://doi.org/10.1145/3559400.3561998

[103] Jasper Tran O'Leary, Chandrakana Nandi, Khang Lee, and Nadya Peek. 2021. Taxon: a Language for Formal Reasoning with Digital Fabrication Machines. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 691–709. https://doi.org/10.1145/3472749.3474779

[104] Nektarios Georgios Tsoutsos, Homer Gamil, and Michail Maniatakos. 2017. Secure 3D Printing: Reconstructing and Validating Solid Geometries using Toolpath Reverse Engineering. In *Proceedings of the 3rd ACM Workshop on Cyber-Physical System Security* (Abu Dhabi, United Arab Emirates) *(CPSS '17)*. Association for Computing Machinery, New York, NY, USA, 15–20. https://doi.org/10.1145/3055186.3055198

[105] UltiCa. 2019. Prusa slicer creates wrong g-code; surface is interpreted the wrong way. https://github.com/prusa3d/PrusaSlicer/issues/3156.

[106] Nobuyuki Umetani and Ryan Schmidt. 2013. Cross-sectional structural analysis for 3D printing optimization. In *SIGGRAPH Asia 2013 Technical Briefs* (Hong Kong, Hong Kong) *(SA '13)*. Association for Computing Machinery, New York, NY, USA, Article 5, 4 pages. https://doi.org/10.1145/2542355.2542361

[107] valely97. 2019. Random missing parts in a 3D Model in Cura 4.0.0. https://community.ultimaker.com/topic/27334-random-missing-parts-in-a-3d-model-in-cura-400/.

[108] VBezchleba. 2019. PREPARE TO PREVIEW - A PART OF MODEL MISSING. https://community.ultimaker.com/topic/28300-prepare-to-preview-a-part-of-model-missing/.
[109] virtuwill. 2024. Model seems different after slicing in a bad way. https://community.ultimaker.com/topic/44793-model-seems-different-after-slicing-in-a-bad-way/#comment-329199.
[110] Visual Computing Lab, ISTI - CNR. 2025. MeshLab: The Open Source System for Processing and Editing 3D Triangular Meshes. https://www.meshlab.net/. Accessed: 2025-03-25.
[111] Voron-0. 2023. A Drive Frame Upper x1. https://github.com/VoronDesign/Voron-0/blob/Voron0.2r1/STLs/A_Drive_Frame_Upper_x1.stl.
[112] Voron-0. 2023. M2 Nut Adapter Rotated x5. https://github.com/VoronDesign/Voron-0/blob/Voron0.2r1/STLs/M2_Nut_Adapter_Rotated_x5.stl.
[113] Voron-0. 2023. Rear Bed Mount Left x1. https://github.com/VoronDesign/Voron-0/blob/Voron0.2r1/STLs/Rear_Bed_Mount_Left_x1.stl.
[114] Voron-0. 2023. Rear Bed Mount Right x1. https://github.com/VoronDesign/Voron-0/blob/Voron0.2r1/STLs/Rear_Bed_Mount_Right_x1.stl.
[115] Voron-0. 2023. Spool Holder x1. https://github.com/VoronDesign/Voron-0/blob/Voron0.2r1/STLs/Spool_Holder_x1.STL.
[116] Voron-0. 2023. T8 Nut Block x1. https://github.com/VoronDesign/Voron-0/blob/Voron0.2r1/STLs/T8_Nut_Block_x1.stl.
[117] Voron Design Team. 2025. Voron 0 CoreXY 3D Printer. https://github.com/VoronDesign/Voron-0. Accessed: 2025-03-22.
[118] Aosen Wang, Tianjiao Wang, Chi Zhou, and Wenyao Xu. 2017. LuBan: Low-Cost and In-Situ Droplet Micro-Sensing for Inkjet 3D Printing Quality Assurance. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems* (Delft, Netherlands) *(SenSys '17)*. Association for Computing Machinery, New York, NY, USA, Article 27, 14 pages. https://doi.org/10.1145/3131672.3131686
[119] werewolftm. 2024. Cura does not see the hole. https://github.com/Ultimaker/Cura/issues/19537.
[120] Wiesiek1952. 2024. Wrong slicing in Windows. https://github.com/prusa3d/PrusaSlicer/issues/12309.
[121] WilmerSuarez. 2024. "Print Thin Walls" keeps failing to slice. https://github.com/Ultimaker/Cura/issues/16726.
[122] Kaushik Yanamandra, Guan Lin Chen, Xianbo Xu, Gary Mac, and Nikhil Gupta. 2020. Reverse engineering of 3D printed composite part by toolpath reconstruction. In *Proceedings of the American Society for Composites - 35th Technical Conference, ASC 2020 (American Society for Composites - 35th Technical Conference, ASC 2020)*, Kishore Pochiraju and Nikhil Gupta (Eds.). DEStech Publications, 195–208.
[123] Zaive. 2020. Fixing bad geometry on a complex mesh. https://blender.stackexchange.com/questions/176139/fixing-bad-geometry-on-a-complex-mesh.
[124] M Šljivic, A Pavlovic, M Kraišnik, and J Ilić. 2019. Comparing the accuracy of 3D slicer software in printed enduse parts. *IOP Conference Series: Materials Science and Engineering* 659, 1 (oct 2019), 012082. https://doi.org/10.1088/1757-899X/659/1/012082

Received 2025-03-25; accepted 2025-08-12