Utrecht University

Faculty of Science
Department of Information and
Computing Sciences

Harpe
Partitioning Models to Minimize the Parallel Print Time in Fused Filament Fabrication

ICA-5785316
October 6, 2021

Utrecht University

Ultimaker

Supervisor:
Prof. Dr. M.J. van Kreveld

2nd Examiner:
Dr. M. Löffner

Author:
C.H.P. Lamboo
`c.h.p.lamboo@students.uu.nl`
`c.lamboo@ultimaker.com`

Daily Supervisors:
Remco Burema
Jaime van Kessel

Abstract

3D printers have become widely accessible. It is increasingly more common for companies, or even individuals to have multiple 3D printers. Having multiple printers allows one to fabricate more objects at the same time, although the earliest time that any single object is available is still bounded by the printing time of a single printer. Partitioning this model such that each sub-model is printed in parallel on a separate printer will greatly improve the print time of the model. HARPE is a model-decomposition algorithm that partitions a 3D model in (at most) n parts such that the print time of the sub-model with the slowest print-time is minimized.

Contents

1 Introduction **1**
1.1 Preliminaries 1
1.1.1 3D manufacturing techniques 1
1.1.2 Fused Filament Fabrication 2
1.1.3 Slicing 2

2 Related Work **4**
2.1 Improve print time 4
2.2 Increase print volume 4
2.3 Avoid support 4
2.4 Improve surface quality 5
2.5 Partitioning a model for packing 6

3 Method **7**
3.1 Definitions 7
3.2 Print Time Estimation 10
3.2.1 Calculating feature volumes 11
3.2.2 Print time estimation 13
3.2.3 Material usage estimate 14
3.2.4 Print Direction 14
3.3 Partition Search 15
3.3.1 Candidate Planes 16
3.3.2 Basic Partition Search 17
3.3.3 Average Cross-section Area Heuristic 19
3.3.4 Support 21
3.4 Batched Calculations 25
3.4.1 Cross-sectional Area 25
3.4.2 Surface Areas 27
3.4.3 Volume 28
3.4.4 Support Volumes' Geometry 31
3.4.5 Support Volumes' Support 33
3.5 Local Search Procedure 34
3.6 Connectors 36

4 Results **39**
4.1 Print Time Estimation 40
4.2 Partition Search 43
4.2.1 Comparison of search strategies 43
4.2.2 Performance of Partition Search 44
4.3 Batched Calculations 45
4.4 Local Search Procedure 46

5 Conclusion **47**

6 Discussion & Future work **48**

7 Acknowledgement **50**

A Partition Examples **51**

1 Introduction

3D printers are used for various applications such as education, manufacturing and prototyping. Prototyping is a process where it is very useful to have a physical representation of the object that is being designed. A common workflow in prototyping is an iterative process where a design is printed. After the design has been printed flaws and possible improvements can be discovered. The design is adjusted and the process starts again. In this process a fast turnaround time is valued. For some models the printing process takes hours or even days. This slow turnaround time stagnates the iterative designing process.

Using multiple 3D printers could solve this issue. The model can be split into a number of parts that is at most equal to the number of available printers. All sub-models can then be printed in parallel. In order for this method to be effective, the model-splitting needs to be automated. Doing this manually would require additional time, defeating the purpose of the high turnaround time. Additionally the assembly process should be as easy as possible. Having a partitioning that requires an intensive assembly process can cost a significant amount of time that exceeds the time gained by the partitioning.

As all parts are printed simultaneously the total print time of all parts is determined by the part with the longest print time. The goal of the algorithm thus becomes an optimization problem of finding the partition G of the input model g containing at most n pieces that minimizes the print time $\texttt{PrintTime(}g_{part}\texttt{)}$ of the slowest printed sub-model $g_{\text{part}} \in G$ (eq. (1)).

$$\underset{\text{partitioning } G \text{ of } g \;:\; |G| \leq n}{\arg\min} \left(\max_{g_{\text{part}} \in G} \left\{ \texttt{PrintTime(}\ g_{part}\ \texttt{)} \right\} \right) \tag{1}$$

This objective is referred to as *minimizing the parallel print time.* Contributions of this work include

- a novel method to estimate the print time, while less precise than previous methods, it can predict the print times of models significantly faster,
- a partition algorithm, that cuts a model in n parts,
- a method calculates properties for a set of candidate cuts, making it possible to evaluate a dense collection of cuts while maintaining an efficient algorithm,
- a local search procedure that improves an existing solution by iteratively optimizing the partitioning, and
- a method for adding connectors between the model-parts for an increased ease of assembly.

1.1 Preliminaries

1.1.1 3D manufacturing techniques

There are many techniques for fabricating 3D models. Examples of such techniques are:

- **Fused Filament Fabrication** (FFF); a form of additive manufacturing where molten filament is extruded through a heated nozzle. The nozzle is guided through a predetermined toolpath,
- **Selective Laser Sintering** (SLS) or **Powder Bed Fusion**; here a model is manufactured by depositing a layer of powder on the whole build area. The area that needs to be solid is fused together using a laser,

- **Milling** (e.g. CNC); a form of subtractive manufacturing, material is removed from a solid block by drilling until only the desired model remains.

Each of these techniques has different characteristics, and the total time spent manufacturing the model is influenced differently by each of these fabrication methods. In this thesis FFF printers will have the main focus.

1.1.2 Fused Filament Fabrication

For most FFF printers 3D models are fabricated by extruding molten plastic through a 3-axis controlled nozzle head. Using this method models can be printed by guiding the nozzle through predetermined tool paths and adding material on the build plate. After the first layer is printed the second layer can be added by using the first layer as the build area. The complete model is fabricated layer by layer, where each previous layer is the build area for the next layer.

1.1.3 Slicing

Before a model can be printed the model needs to be converted to instructions the printer can understand: G-Code. The process of generating these print instructions is called *slicing*. Slicing is usually done using slicing software such as CURA[23] (fig. 1).

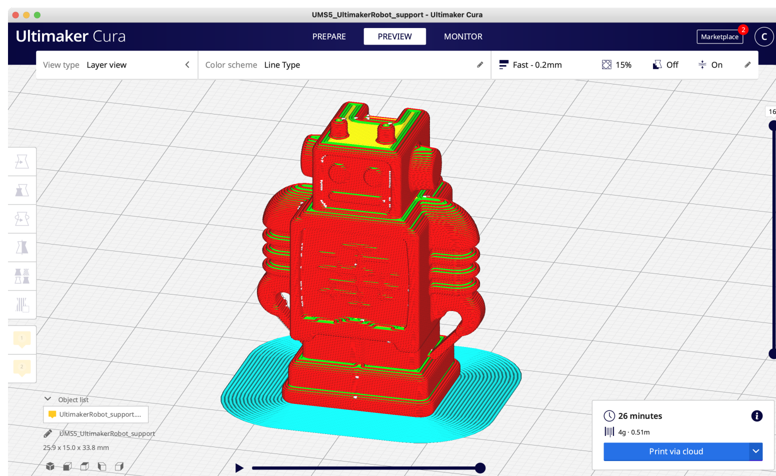
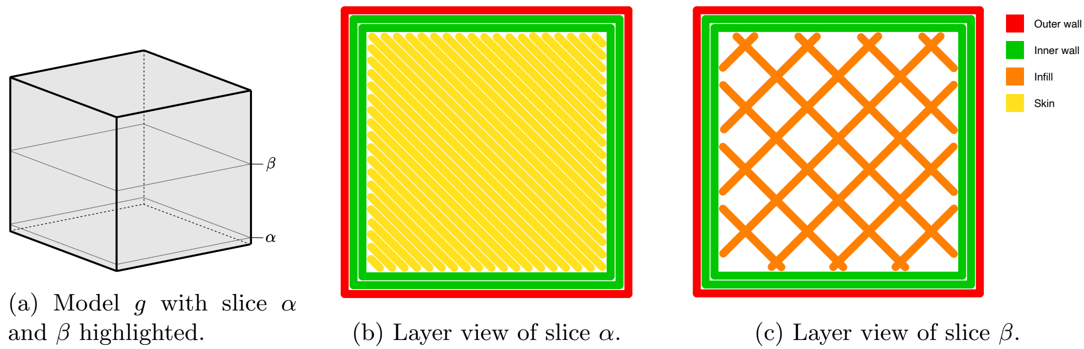


Figure 1: Screenshot of slicing software CURA showing a preview of the tool paths.

The slicing process splits a model into layers. Each layer fills the two-dimensional polygon shape resulting from the cross section between a horizontal plane and the target model. Each layer is built from a collection of tool paths. This is highlighted in fig. 2. Figure 2a shows the original target model with layers α and β highlighted. Tool paths for these layers are shown in fig. 2b and fig. 2c.



(a) Model g with slice α and β highlighted.

(b) Layer view of slice α.

(c) Layer view of slice β.

Figure 2: Visualisation of the tool paths for a cube model.

These tool paths can all be assigned to a *feature*, each feature serves a different purpose and is printed with different characteristics. The main features used by CURA are.

- a slowly printed **outer wall** that is visible on the outside,
- an **inner wall**, which is printed faster,
- a **skin** that fills the outer shell of the model that is not covered by the wall,
- a coarsely printed **infill** that provides structural integrity on the inside of a model, and
- a **support** structure, as it is not possible to print on top of a non-existent previous layer.

The slicing process can be customized through slice settings. A simplified version of the slice settings used in CURA is shown in listing 1. Changing these slice settings will influence the print time and material usage. Throughout the thesis this `deviating` font type is used to refer to these settings.

```
Settings:
    material_diameter:      mm
    nozzle_size:            mm
    layer_height:           mm
    wall_thickness:         mm
    top_thickness:          mm
    bottom_thickness:       mm
    infill_percentage:      percentage
    support_percentage:     percentage
    inner_wall_speed:       mm/s
    outer_wall_speed:       mm/s
    skin_speed:             mm/s
    support_speed:          mm/s
    infill_speed:           mm/s
```

Listing 1: Relevant slice setting values with their corresponding physical units.

2 Related Work

A lot of research has been conducted in the field of the decomposition of 3D models. There are several motivations for partitioning a model, such as improving print time (section 2.1), partitioning the model so its volume can be increased (section 2.2), removing the need for support (section 2.3), improving the surface quality (section 2.4), and packing (section 2.5).

2.1 Improve print time

A moderate amount of research has been done where models are decomposed with the goal to speed up the print process.

Both Chen et al.[16] and Chen et al.[11] propose a method where only the shell of a model is printed in pieces. Both approaches have an inner structure that is constructed separately from the print process. For [16] this inner structure is made from pre-manufactured building blocks to which the printed outer shell pieces are connected. In [11] an improvised skeleton of the model is constructed using crafting material. The shell-pieces are assembled by gluing the pieces to this skeleton.

2.2 Increase print volume

The sizes of 3D printed models are bounded by the build-volume of the printer. One way to circumvent this restriction is to partition a model such that each partition is not bigger than the build-volume. The assembled parts can then be many times the size of the original build volume.

Chopper[6], an algorithm designed by Luo et al., uses beam search to find a BSP (Binary Space Partition) of the input model. Their objective criteria are not limited to make sure that every part fits within the build-volume, but also take into account the assemblability[1] of the parts, structural soundness of the assembled model, and aesthetics. An elaborate connector placing scheme is designed to simplify the assemblability process, and makes for a sturdier assembled model. The objective of Chopper is highly modular and could easily be adjusted for different needs.

Song et al. place a model in a voxel grid in [10]. The model partitioning is based on those voxel cells that are covered by (part) of the model. Voxels are grouped together to form parts that can be assembled without glue as the pieces are self-interlocking.

Jiang et al. shrink the input model to a 0-volume skeletal representation of the original mesh[13]. Each position on the skeleton corresponds to a part of the surface on the original model. Parts of the skeleton that correspond to a minimum local surface area are good candidate cuts. Consequently, the model is partitioned on these locations.

2.3 Avoid support

Using FFF-printers, overhangs are harder to print. These printers construct a model by fusing additional material to an already printed part of the model. This is usually done by splitting the model into layers, printing the model layer by layer. In order to get a successful print, each layer needs support from the previous layer(s). For some models the model itself does not provide for sufficient support, so a support structure is printed along the model itself for the needed support. These support structures are not part of the model, and thus need to be removed after printing. These structures require additional material, and removing these structures is manual work that may damage the model and leave marks on the final product. Considerable work has been done

[1] A term introduced by Chopper, meaning whether it is possible to assemble the final model

to partition the 3D model to eliminate, or minimize, the overhanging areas and thus the need for these support structures.

Hu et al. aim to solve the k-pyramidal shape decomposition problem in [7]. A shape is considered pyramidal if it has a flat base, and is x-monotone in the remainder of its shape. As the k-pyramidal shape decomposition is a very complex problem, the shapes produced by the algorithm are not strict, but approximate pyramidal shapes. The algorithm works by sampling the interior of the input shape. A number of planar cuts are considered. A planar cut ℓ is the full, infinite plane defined by a normal $\overrightarrow{n}$ and a distance d. A sample point v is said to be covered by a planar cut ℓ, if the minimal distance from the sample location v to the planar cut ℓ does not travel outside the input shape. A set number of cut-directions $\overrightarrow{n}$ are generated, the number of which is controlled by a parameter. Each of the sample locations *vote* for a number of cuts. A cut is voted for if the sample location is covered by the aforementioned cut. Neighbouring sample locations that vote for the same bases are clustered together. Each cluster has multiple valid cuts that could be used as print base, and each cluster is (approximately) pyramidal. Then neighbouring clusters are merged by solving an exact cover problem over the candidate parts to obtain the final pyramidal decomposition.

In [18] an object is partitioned by recursively applying three decomposition tactics. One of the tactics is *base extraction*. Here a large flat base is identified. The largest possible portion of the model that can be printed using this flat area as a print base is found and cut from the model. The second tactic is *tip extraction*. Narrow features that are not likely to be printed as a print base are identified. From such features a region is grown until a large enough print base can be identified. The print base then serves as a cut to partition part of the model. Finally T-Junctions, two parts of the model that are connected perpendicularly, are identified using a voxel approach. The two parts are split using a planar cut. These three approaches are applied recursively, on each recursion step one approach is chosen with a random probability. This recursive process is repeated until each piece is assigned a print direction, and can be printed without support.

Yu et al. employ evolutionary computing to find a BSP over the input model[15]. The objective of the genetic algorithm is to find a partition that minimizes the total overhang area, while also trying to keep the part-count as low as possible. Both the Multi-objective Generic Algorithm (MOGA) and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) strategies are used to find the decomposition.

Wei et al. start their approach similarly to the approach described in [13] by creating a 0-volume skeletal representation of the model using Laplacian smoothing[17]. As a result from the Laplacian skeletonization the surface area, and the skeletal edge that corresponds to this surface area skeleton, are approximately perpendicular to each other. A section of the model can be printed without support if all of its skeletal edges have an orientation that is at least equal to the maximum overhang angle of the printer. Wei et al. solve the decomposition problem by decomposing the skeleton such that each part fits within a cone with an angle that is twice the maximum overhang angle.

2.4 Improve surface quality

For FFF printers a model is usually fabricated by slicing a model into layers. The resolution in the layer direction (usually the z-axis) is far less than the resolution in the two other directions. For models where surface quality is of importance, the surfaces with a large amount of detail need to be printed perpendicular to the print direction. Partitioning a model might help finding the ideal print direction for a larger portion of the surface of the model.

In [12] Wang, Zanni, and Kobbelt aim to solve this problem by assigning each face to a set

of candidate print directions. Adjacent faces are grouped together to find a balance between number of surface patches, and local ideal print direction. Then the model is partitioned by finding a Voronoi region that approximates these surface patches. The intersection between a Voronoi region and the model form a partition.

Filoscia et al. aim not only to print the surfaces in the ideal print direction, but also eliminate the need for any support on the surface of the model[19]. First, patches with similar surface characteristics are found. These patches are then grouped together using Linear Programming (LP). The objective of the LP is not only to find a balance between the number of patches and the ideal print direction, but also to prevent the need for support on any of these surfaces. The result is a partition on the surface of the model where the ideal cuts would be. These cuts are then slightly moved to creases in the model to make the cuts less apparent. The creases are found by using the ambient occlusion of the model; places where less light can penetrate are less visible, and are good candidate places where these seams can be hidden.

2.5 Partitioning a model for packing

Packing algorithms aim to decompose an object, and then tightly pack the parts. This kind of algorithms are most applicable for SLS printers. For SLS printers each layer is completely filled by a powder, the model is constructed by fusing parts of the layer that belong to the model. The powder that is not part of the model is discarded and *cannot* be used again. In order to minimize waste, model (parts) are tightly packed together. Miscellaneous research has been done on how to decompose a model, and pack the pieces together such that the height of the packed components is minimized. SLS printers will more reliably print overhang areas compared to FFF printers. Because of this the packing of objects may contain intricate overhang areas that are almost impossible to print for FFF printers.

Chen et al. use the results from [7] to create an initial pyramidal decomposition of the model [9]. These parts are then voxelized, using the pyramid-base adjacent to the voxel boundary. These pieces are then iteratively added to the packed build-volume. Parts may be further partitioned during this process through axial cuts. Using beam search multiple configurations of the packed build-volume are considered during the packing process. Only the most promising configurations are explored in the search tree.

In "PackMerger: A 3D Print Volume Optimizer" Vanek et al. partition a model by first extracting the shell of the input model[8]. This shell is converted into a set of tetrahedral cells and split into a large number of segments. These are combined using k-means clustering to create an initial partitioning of the model. The relation between the pieces is stored in a weighted undirected graph, with edge weights equal to the cross-sectional area between the segments. Pieces with little contact area are hard to assemble, so these pieces are merged. Tabu search and gradient descent are used to tightly pack the pieces in a minimum-volume bounding box by changing the orientation and position of the pieces.

3 Method

Recall that the goal of this research is to develop an algorithm that computes a partition G of the input model g containing at most n pieces that minimizes the print time $\texttt{PrintTime(}\ g_{part}\texttt{)}$ of the slowest printed sub-model $g_{\text{part}} \in G$.

$$\underset{\text{partitioning } G \text{ of } g \ : \ |G| \leq n}{\arg\min} \left(\max_{g_{\text{part}} \in G} \{\texttt{PrintTime(}\ g_{part}\ \texttt{)}\} \right)$$

A valid partitioning G must comply with the following properties.

- The number of model-parts is at most equal to the number of available printers n

$$|G| \leq n \tag{2}$$

- When assembling the parts the result should be the original model. The union of all geometry-parts results in g

$$\bigcup_{g_{\text{part}} \in G} g_{\text{part}} = g \tag{3}$$

- There is no overlap between geometry-parts

$$\left\{ g_a \cap g_b \mid g_a, g_b \in G, g_a \neq g_b \right\} = \varnothing \tag{4}$$

In order to compare partitions, the print time $\texttt{PrintTime(}g\texttt{)}$ for each piece needs to be known. Previously the best method for doing this is slicing the model to generate the tool paths. After these tool paths are generated, the print process is simulated by traversing all tool paths. This will be too time consuming as the search-algorithm will need to know the print time for a large number of model pieces. Section 3.2 describes a novel method for estimating the print time more efficiently. While less accurate the increase in performance makes this new approach useable for the algorithm. In section 3.3 the search method for finding the partitioning is described. The *min-max* print time goal for this thesis is achieved using a greedy heuristic search procedure that at each step of the algorithm evaluates a number of candidate cuts. Evaluating a higher number of candidate cuts results in a better partitioning. Section 3.4 describes a method to efficiently evaluate these candidate cuts. After a partitioning has been found the partitioning is improved by iteratively updating the partitioning in section 3.5. Finally connector pieces are added to the partition surfaces for an improved assembly process (section 3.6).

3.1 Definitions

During the explanation of the algorithm there are recurring geometrical objects. In the following section the most important objects are described.

Geometry Models are represented as a geometry g, an unordered triangle list. Geometry g contains m triangles $g = t_1, t_2, \ldots, t_m$, and each triangle t_i consists of three points $t = (t_a, t_b, t_c)$.

As these geometries represent solid models we need to have a well-defined in and outside of the geometry. When directly facing the triangle the side of the triangle is said to be on the outside if the order of points is clockwise. Alternatively if the order of points is counter clockwise when directly facing the triangle that side is on the inside.

The geometries used in this thesis are limited to those models that are *manifold*; every triangle edge has exactly one other incident triangle and there are no overlapping/intersecting faces.

Hyperplane The hyperplane is a geometric element for representing planes, generalized to any dimension d. Any point $(v_1, v_2, \ldots, v_d)$ for which eq. (5) holds is located on the hyperplane.

$$\mathsf{HyperPlane}^d := \{x_1 v_1 + x_2 v_2 + \cdots + x_d v_d = 0\} \tag{5}$$

The most common hyperplane variant used in this thesis is the regular plane and can be expressed as $\mathsf{HyperPlane}^3$ otherwise referred to as p. The two-dimensional variant (a line) can be expressed as $\mathsf{HyperPlane}^2$ or ℓ.

A $\mathsf{HyperPlane}^3$ can be used to partition a geometry g, cutting it into a top geometry $g^\top$, and a bottom $g^\perp$ geometry ($\texttt{Partition}(p,\ g) \mapsto (g, g)$). The top part $g^\top$ is the geometry from g that is above p, while bottom part $g^\perp$ is the geometry that is below p. This is illustrated in fig. 3.

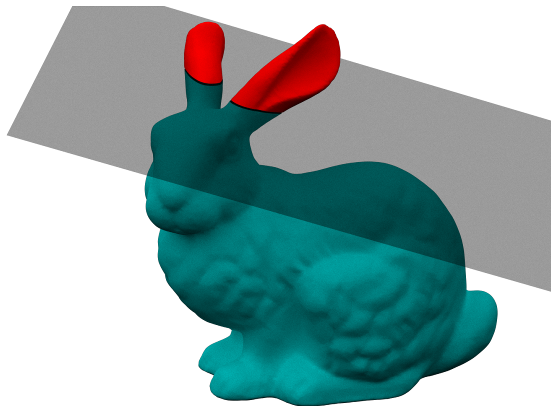


Figure 3: A plane p partitioning a geometry g into geometries $g^\top$ (red) and $g^\perp$ (turquoise).

Partitioning a geometry might cut a geometry in disjoint pieces, however these disjoint pieces are considered to be a single geometry. This is done to provide more control over the cuts; using this definition of a cut, partitioning a geometry always results in exactly two geometries.

Parallel Planes An ordered list of k parallel planes $P = p_1, p_2, \ldots, p_k$ is frequently used throughout the algorithm. During the search procedure a batch of candidate partition-planes are evaluated before picking a single plane to perform a cut. The space in between each successive plane is constant throughout the batch, τ is used to denote this distance in millimetres. Figure 4 illustrates such a set of parallel planes for the Lucy model.

Figure 4: A set of parallel planes P with accompanied geometry g.

Various properties can be calculated on a geometry g in combination with a collection of parallel planes P. Examples of such properties used in this work include:

- `CrossSectionalAreas(`P, g`)` $\mapsto [\mathbb{R}]$; for each plane $p \in P$ calculate the area of the cross section with geometry g,
- `Areas(`P, g`)` $\mapsto ([\mathbb{R}], [\mathbb{R}])$; for each plane $p \in P$ calculate the area of $g^\top$ and $g^\perp$ if geometry g were to be partitioned by p, and
- `Volumes(`P, g`)` $\mapsto ([\mathbb{R}], [\mathbb{R}])$; for each plane $p \in P$ calculate the volume of $g^\top$ and $g^\perp$ if geometry g were to be partitioned by p.

Later in section 3.4 we will show that these operations are calculated efficiently. Having these efficient operations allow for a higher density of candidate partition-planes.

Half-space Similar as the hyperplane, the difference is that the half-space also includes the region below the plane (when oriented in the direction of the half spaces' normal). Any point $(v_1, v_2, \ldots, v_d)$ for which eq. (6) holds is contained within the half-space.

$$\mathsf{HalfSpace}^d := \{x_1v_1 + x_2v_2 + \cdots + x_dv_d \leq 0\} \tag{6}$$

There is not a variant of the half-space where the sign is flipped; this is redundant as it can be achieved by negating the half-space.

Binary Tree A tree is a recursive data structure that can be either one of two elements; a *node* (depicted as a circle in fig. 5b) or a *leaf* (depicted as a square in fig. 5b). The nodes contain a left and a right child that are themselves instances of the tree data structure. A location in a tree can be expressed through a sequence of *left* and *right* turns from the tree's root. Data can be stored in both the *leaf* and *node* elements of the binary tree. For our definition of a binary

tree we borrow a concept from functional programming, namely *algebraic data types*. Such data types contain variables in their definition, denoting the type of data contained within the data type. For our case these variables are the data types for the *leafs* and *nodes*.

$$\mathsf{Tree}\langle leaf,\ node\rangle$$

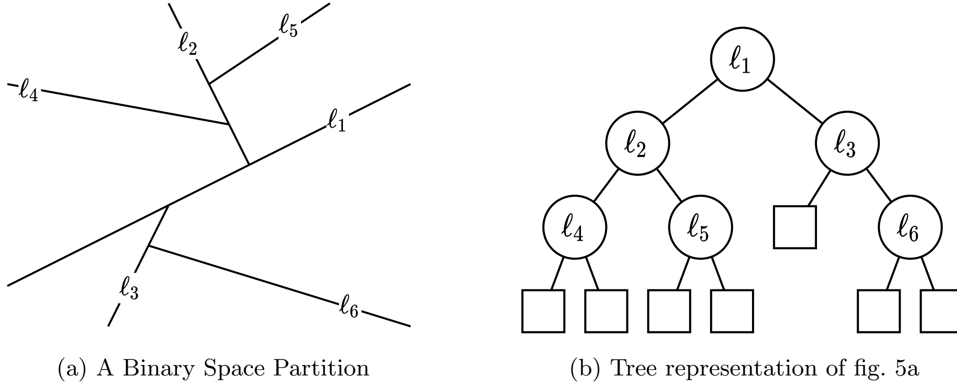
For instance, a $\mathsf{Tree}\langle \mathbb{N},\ \ell\rangle$ corresponds to a binary tree that stores lines at the nodes and a natural number in the leaves.

Observation 1. *A tree-branch can only terminate in a leaf.*

Lemma 2. *In all non-empty trees of finite depth there is at least one node that has a terminating leaf for both the left and right child.*

Proof. By contradiction. Suppose that this would not be the case then all nodes should have (at least) one child that is also a node. Such a node does not result in a termination. As this is the case for all nodes, the tree cannot terminate and we thus have a tree of infinite depth. As the lemma states we have a tree of finite depth, we have found a contradiction. □

BSP A Binary Space Partition (BSP) is a tree data structure that partitions space using hyperplanes. Each node of the tree corresponds to a hyperplane. The sub tree on the left child of this node corresponds to the space above p, and the right child corresponds to the space below p. Each node recursively splits more space in a similar fashion. A simple BSP is illustrated in fig. 5a, the tree structure of the same BSP is depicted in fig. 5b.



(a) A Binary Space Partition

(b) Tree representation of fig. 5a

Figure 5: A BSP partitioning the plane (fig. 5a) and the tree representation of the same BSP (fig. 5b).

For our definition of a BSP the BSP is defined in terms of a binary tree. Note that the unit type () is used to denote that a BSP does not contain any data in the leaves.

$$\mathsf{BSP}^d := \mathsf{Tree}\langle (),\ \mathsf{HyperPlane}^d\rangle$$

3.2 Print Time Estimation

The time spent printing a model, `PrintTime(`g`)`, is an important metric throughout Harpe. Using slicing software Cura the print time can be precisely calculated. The print time can be calculated by first generating all tool paths for a model. After the tool paths are created the

print time, $\texttt{PrintTime}_{\texttt{CURA}}(g)$, is calculated by traversing all tool paths and for each tool path calculating the time spent printing. Unfortunately this method is computationally too expensive. Instead, a novel method for calculating the print time is proposed; $\texttt{PrintTime}_{\texttt{HARPE}}(g)$. As this method is used with high frequency to evaluate large sets of candidate partitions this method should be very efficient; any program with a running time worse than linear cannot be used.

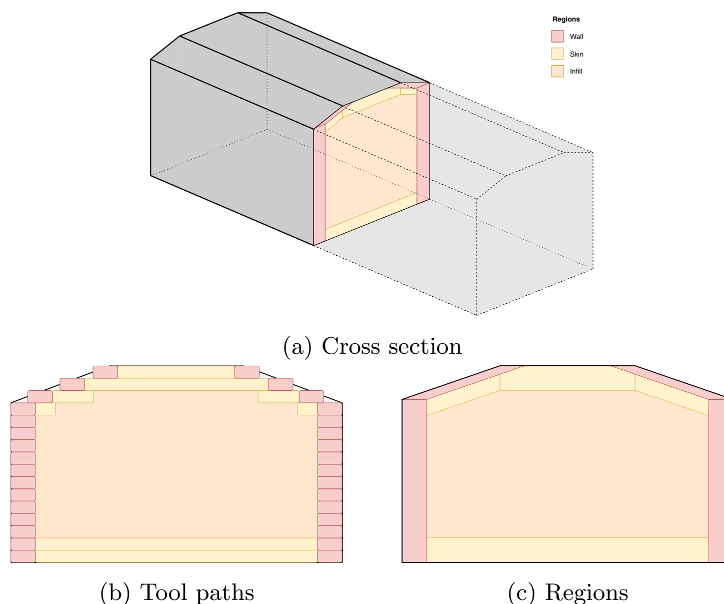
The method for estimating the print time for a model g works by calculating the time spent on each feature separately. For a given feature the process of determining the print time is split into two steps. First the volume for each feature $v_{\text{feature}}(g)$ in mm^3 is calculated. Additionally the print speed f_{feature}, denoted as flow rate in mm^3s^{-1}, for each feature is determined based on the print settings. Then the print time of each feature is its volume divided by its print speed. By looking at the physical units this makes sense; $\frac{mm^3}{mm^3s^{-1}} = s$. The total print time is the sum of each volume's print time. This is shown in eq. (7).

$$\texttt{PrintTime}_{\texttt{HARPE}}(g) = \sum_{\text{feature} \in \{\texttt{inner wall}, \texttt{outer wall}, \texttt{skin}, \texttt{infill}, \texttt{support}\}} \frac{v_{\text{feature}}(g)}{f_{\text{feature}}} \tag{7}$$

3.2.1 Calculating feature volumes

The regions occupied by each feature are traditionally calculated by processing each layer individually. For HARPE the model g is never split into layers. As we still need to know the volume occupied by each feature a different approach for calculating these regions is proposed. Figure 6 illustrates the difference between the tool paths and regions.

For our print time estimation all triangles t in geometry g are traversed. For each triangle t the contribution of triangle t to each feature is determined. The calculations are not exact as other parts of the geometry might intersect a features' volume. However, considering these special cases would make the algorithm too slow and unfit for our purpose.



(a) Cross section

(b) Tool paths

(c) Regions

Figure 6: Cross section (fig. 6a) of geometry g with tool paths (fig. 6b) and regions (fig. 6c) for the *wall*, *skin* and *infill* features highlighted.

Wall volume For a given triangle t the surface area does **not** directly determine the wall volume, but instead the *horizontal projected area* of t is used. This is because the volume that is filled by the wall feature is determined by the horizontal distance to a boundary of model g; a face that is oriented horizontally will not contribute to the total wall volume, while a face that is oriented vertically does. Let `Normal(`t`)` be the surface normal of unit length for triangle t. The *horizontally projected area* of triangle t is calculated by taking the horizontal vector length of normal $|\texttt{Normal(}t\texttt{)}_{xy}|$ multiplied by the area of t. In order to get the wall volume, the horizontally projected area is multiplied by the `wall thickness` eq. (8).

$$\texttt{HorizontallyProjectedVolume}(t) = |\texttt{Normal(}t\texttt{)}_{xy}| \cdot \texttt{Area(}t\texttt{)} \cdot \texttt{wall_thickness} \tag{8}$$

Then the total wall volume $v_{\text{wall}}(g)$ of model g is approximately the sum of all triangle wall volumes eq. (9). This is approximate as narrow features in a model might cause the wall regions of two separate triangles to overlap. For the exact volume these overlapping regions should only be calculated once.

$$v_{\text{wall}}(g) = \sum_{t \in g} \texttt{HorizontallyProjectedVolume}(t) \tag{9}$$

Note that the wall volume is split into an *inner wall volume*, and an *outer wall volume*. In order to correctly calculate these volumes the `wall thickness` in eq. (8) is replaced

- by `nozzle_size` for the outer wall as the outer wall is a single extrusion line on the outer contour of the layer,
- and by `wall_thickness` − `nozzle_size` for the inner wall, as the remaining volume of the wall that is not filled by the outer wall is filled using the inner wall feature.

Skin volume Similarly to the wall volume discussed previously, the skin volume is determined by first calculating the *top projected area* for each triangle t. This *top projected area* is then multiplied by the `SkinThickness` eq. (11). As the bottom and top thickness can be modified separately the `SkinThickness` is equal to either `top thickness` or `bottom thickness` depending on the surface normal of triangle t. If the surface normal is pointing up then triangle t is part of the top surface thus `top thickness` is used. Otherwise `bottom thickness` is used as the skin thickness eq. (12).

The total skin volume $v_{\text{skin}}(g)$ is then determined by the sum of all skin volumes of all triangles. However, some regions within the model might be assigned to both the skin and wall features. If this is the case then the wall feature takes precedence over the skin feature. Let r_{feature} be the region assigned to each feature. Then the correct volume for the skin would be $r_{\text{skin}} - r_{\text{skin} \cap \text{wall}}$. Calculating the intersection between these regions would be too costly, thus this intersection is calculated on a per-face bases, where the skin region is considered to be a subset of the outer wall region. The resulting value is capped at 0 to prevent negative volumes (eq. (10)).

$$v_{\text{skin}}(g) = \sum_{t \in g} \max\Big\{0, \texttt{HorizontallyProjectedVolume}(t) - \texttt{TopProjectedVolume(}t\texttt{)}\Big\} \tag{10}$$

$$\texttt{TopProjectedVolume(}t\texttt{)} = |\texttt{Normal(}t\texttt{)}_{z}| \cdot \texttt{Area(}t\texttt{)} \cdot \texttt{SkinThickness(}t\texttt{)} \tag{11}$$

$$\texttt{SkinThickness(}t\texttt{)} = \begin{cases} \texttt{top_thickness} & \text{if } \texttt{Normal(}t\texttt{)}_{z} > 0, \\ \texttt{bottom_thickness} & \text{otherwise.} \end{cases} \tag{12}$$

Infill volume The remaining volume of model g that is not filled by the skin, inner wall, and outer wall volumes is printed by the infill. As such the infill volume $v_{\text{infill}}(g)$ is calculated by subtracting the skin, inner wall and outer wall volumes from the total volume $v_{\text{total}}(g)$ of model g. The total volume $v_{\text{total}}(g)$ of model g is calculated using the *sum of signed volumes* as described in [2] and in detail explained in section 3.4.3. As the infill volume is not printed solid, but as a coarse structure, the calculated infill volume is multiplied by the `infill percentage`.

$$v_{\text{infill}}(g) = (v_{\text{total}}(g) - v_{\text{skin}}(g) - v_{\text{inner wall}}(g) - v_{\text{outer wall}}(g)) \cdot \texttt{infill_percentage} \tag{13}$$

Support volume For each downwards oriented triangle support is needed. The support structure is considered to be a solid triangular column that spans all the way from the print base to the triangle. The total support volume is the sum of all these triangular columns, eq. (14).

$$v_{\text{support}}(g) = \sum_{t \in g} \begin{cases} \texttt{TriangularVolume}(t) & \text{if } \texttt{Normal}(t)_z < 0, \\ 0 & \text{otherwise.} \end{cases} \tag{14}$$

3.2.2 Print time estimation

After the volumes for each feature has been calculated the print time can be derived. First the print speed is determined. This is the rate at which filament is extruded through the nozzle; the flow rate. Flow rate is denoted in mm^3s^{-1}. Lines extruded by the print head have a height equal to the `layer height`, and the width of the line is determined by the `nozzle size`. Let $\texttt{speed}_{\text{feature}}$ be the print speed of each of the features (i.e. `skin_speed`, `outer_wall_speed` etc). As the print head moves with $\texttt{speed}_{\text{feature}}$ millimetres per second, the length of a line extruded in a single second is equal to $\texttt{speed}_{\text{feature}}$. For each feature the volume extruded in a single second, the flow rate, is f_{feature} as described in eq. (15).

$$f_{\text{feature}} = \texttt{nozzle_size} \cdot \texttt{layer_height} \cdot \texttt{speed}_{\text{feature}} \tag{15}$$

As previously discussed the total print time is calculated by dividing the volume by the print speed (eq. (7)). Pseudo code for finding the volumes for each feature, and calculating the total print time is shown in Algorithm 1. The time complexity for this method is dictated by the loop going through all m triangles. All operations within this loop are executed in constant time. We will later show in section 3.4.1 that volume calculated in line 15 has a time complexity linear in the number of triangles m. Sequencing these operations results in a time complexity of $O(m)$.

Algorithm 1: Calculating print time for model g

```
   Input  : Geometry g and the print settings
   Output: Estimate of the print time in seconds
1  Function PrintTime_HARPE(g) is
2      skinVolume, innerWallVolume, outerWallVolume, supportVolume ← 0
3      foreach t ∈ g do
4          Let n⃗ be the normal of unit length of triangle t.
5          skinThickness ← { top_thickness     if n_z > 0,
                             bottom_thickness  otherwise.
6          skinArea ← Area(t) · |n_z|
7          wallArea ← Area(t) · |n_xy|
8          outerWallVolume += wallArea · nozzle_size
9          innerWallVolume += max{0, wallArea · (wall_thickness − nozzle_size)}
10         skinVolume += max{0, skinArea · skinThickness − wallArea · wall_thickness}
11         if n_z < 0 then
12             supportVolume += TriangularVolume(t)
13         end
14     end
15     infillVolume ← (Volume(g) − skinVolume − innerWallVolume − outerWallVolume) ·
        infill_percentage
16     supportVolume ← supportVolume · support_percentage
17     f_feature ← nozzle_size · layer_height · speed_feature
18     return skinVolume/f_skin + innerWallVolume/f_inner wall + outerWallVolume/f_outer wall + supportVolume/f_support + infillVolume/f_infill
19 end
```

3.2.3 Material usage estimate

Using this approach the material usage for a geometry can be predicted. While not relevant to the remainder of the thesis it shows the versatility of the approach.

Finding the amount of material needed is fairly easy once we know the volumes for each feature. In FFF printers the material is provided through spools of plastic called filament. Filament is a long cylindrical shaped strand of plastic. The material usage of a model g is defined as the length λ of filament required. The volume of plastic in a strand of length λ is equal to $\lambda\pi(\frac{\texttt{material_diameter}}{2})^2$. The total volume of plastic that is inside the model is equal to the volume of filament required. Solving for λ provides for the formula for calculating the length of plastic needed from the spool (eq. (16)).

$$\begin{aligned} \lambda\pi\left(\frac{\texttt{material_diameter}}{2}\right)^2 &= \sum v_{\text{feature}}(g) \\ \lambda &= \frac{\sum v_{\text{feature}}(g)}{\pi(\frac{\texttt{material_diameter}}{2})^2} \end{aligned} \tag{16}$$

3.2.4 Print Direction

The algorithm described previously in section 3.2.2 has a dependency on the print direction. This is due to the separation of the *wall*, *bottom*, and *top* features. For most practical print

settings these three features contain the same settings for thickness and print speed. If this is the case print direction does not influence the print time (not taking support into account).

If the previous paragraph feels counter intuitive, imagine the following: changing the orientation does not change the print speed, area or volume. If the same volume is printed with the same print speed then the print times will be equal. The only difference in print time can be attributed to how optimized the travel paths are in each orientation. Different print orientations might change the height of a model. One could argue that, for models that become taller when changing orientation, more time is spent moving the print head from bottom of the print bed to the top of this model. While this is true virtually no time is spent changing z position while moving to the next layer; the z travel time is negligible on the whole print time.

When making the print time direction independent the settings `wall thickness`, `bottom thickness` and `top thickness` and `wall speed`, `bottom speed` and `top speed` need to have equal values. If this is the case then these are combined into settings `shell thickness` and `shell speed`. The `shell volume` is calculated by multiplying the `shell thickness` by the area of g. The time it takes to print the shell is calculated similar to the other features; the flow rate f_{shell} is determined making the print time equal to $\frac{v_{\text{shell}}}{f_{\text{shell}}}$.

Pseudo code for this *simpler* print time estimation is given in Algorithm 2. This method runs in constant time, given that the *volume*, *area* and *support volume* have already been calculated.

Algorithm 2: Calculating print time for model g

Input : Area in mm^2, volume in mm^3 and support volume in mm^3 of geometry g.
Output: Estimate of the print time in seconds

```
1 Function PrintTime_{HARPE^Ω}(area, volume, supportVolume) is
2     innerWallVolume ← area · nozzle_size
3     outerWallVolume ← area · (wall_thickness − nozzle_size)
4     supportVolume ← supportVolume · support_percentage
5     infillVolume ← (volume − wall_thickness · area) · infill_percentage
6     f_feature ← nozzle_size · layer_height · speed_feature
7     return ← innerWallVolume / f_inner wall + outerWallVolume / f_outer wall + supportVolume / f_support + infillVolume / f_infill
8 end
```

3.3 Partition Search

Partition Search is a recursive model decomposition algorithm that partitions an input geometry such that the maximum print time of all geometry-parts is minimized. A large number of candidate cuts are heuristically evaluated to find the locally optimum partition plane. To do this the print time is estimated as described in section 3.2.

The results of the partitioning algorithm is BSP-tree. This BSP-tree can be used to partition geometry g; the intersection between the space of a BSP-leaf and g forms a partition-part. There are multiple reasons for using a BSP-tree. Models partitioned using a BSP have flat surfaces where the model is partitioned. These flat surfaces can function as the print base for the model-part. Additionally when a model is partitioned using a BSP it is always possible to assemble the pieces. Later in section 3.6 we will discuss how connectors will be added to the model-parts. These connectors are pins oriented in the same direction as the normal of the partition-plane. The only way for two pieces to be assembled is sliding the pieces together in the direction of the pin.

Observation 3. *Let $g^\top$ and $g^\perp$ be a partitioning of geometry g using any plane p. Parts $g^\top$ and $g^\perp$ can always be assembled by sliding them together in the direction of the surface normal*

of plane p.

Lemma 4. *There always exists an ordering on connecting the pieces of a model partitioned using a BSP such that none of the connectors prevent the pieces from being assembled further.*

Proof. By lemma 2 we have that all trees of finite depth should have at least one node with two leaves as children. For the case of a BSP such a node corresponds to a partition-plane and two geometry-parts. From observation 3 we know that such a partitioning can always be assembled. Connecting these pieces results in a single geometry and can be considered as a single model-part, or a new leaf in the BSP tree.

As this operation decreases the total number of nodes in the BSP tree and for non-empty trees it is always possible to perform this operation we can repeat this operation until we are left with an empty tree. Such a tree corresponds to a fully assembled model. □

The number of partition-pieces cannot exceed n, the number of available printers. As it turns out, for an optimal parallel print time partition, the partitioning will always consist of exactly n pieces. This follows from the print time analysis; let g be an arbitrary model and let the print time be defined as

$$\texttt{PrintTime}(g) = \alpha \cdot \texttt{Area}(g) + \beta \cdot \texttt{Volume}(g)$$

α, and β are some positive constants that follow from the print-time analysis. Let $g^\top$ and $g^\perp$ be the resulting parts for any plane p partitioning g.

Lemma 5. *The print time of* $\texttt{PrintTime}(g^\top)$ *and* $\texttt{PrintTime}(g^\perp)$ *must be less than the print time of* $\texttt{PrintTime}(g)$.

Proof. Let us first consider part $g^\top$. The volume of $g^\top$ is strictly smaller than g as $g^\top$ is a strict subset of g. The surface area that is removed by the partitioning was non-flat. As the cut that replaces this surface is now flat, the overall area is now decreased.

$$\begin{aligned} \texttt{PrintTime}(g) &= \alpha \cdot \texttt{Area}(g) + \beta \cdot \texttt{Volume}(g) \\ &> \alpha \cdot \texttt{Area}(g^\top) + \beta \cdot \texttt{Volume}(g^\top) \\ &= \texttt{PrintTime}(g^\top) \end{aligned} \tag{17}$$

The same argument can be made that the print time $\texttt{PrintTime}(g^\perp)$ must be less than $\texttt{PrintTime}(g)$. □

It is thus beneficial for a parallel print time partitioning to always partition in n pieces. If the partitioning consists of $< n$ pieces the part with the longest print time in the partitioning can be split for a better solution.

3.3.1 Candidate Planes

To find the partition plane p at each recursion step of the algorithm a number of candidate partition-planes are evaluated. These candidate partition-planes need a surface normal orientation. For an effective search method these normals need to be uniformly distributed, as the best partition-plane can be oriented in any direction. Additionally the density of these normals should be balanced. By having too few candidate partition planes we might overlook good cuts while having too many impairs the run time of the algorithm.

The vectors used as surface normals for the planes are generated using a *subdivided octahedron*. The octahedron is an eight-sided shape, and is defined using six vertices (see fig. 7). Each face of the octahedron is a triangular polygon. The six vertices of the octahedron are located at unit length directed in both the positive and negative directions for each of the orthogonal axis directions.

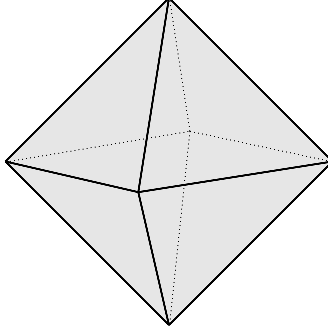
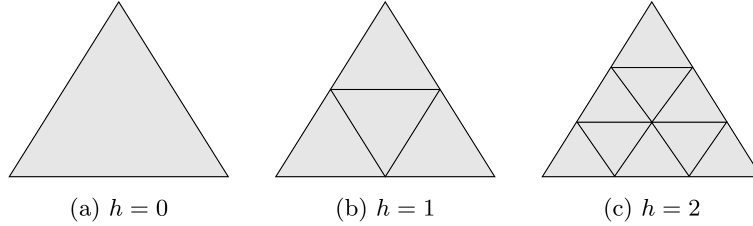


Figure 7: The octahedron shape.

To generate the normals each triangle is subdivided h times. The faces after $h = 0, 1, 2$ subdivisions are shown in fig. 8. Before these vertices are used as normal vector they are normalized to unit length. The vertices of a trice subdivided octahedron are normally used when generating normals throughout the algorithm.



(a) $h = 0$ (b) $h = 1$ (c) $h = 2$

Figure 8: A $h = 0, 1, 2$ subdivided face.

The number of vertices in a subdivided triangle are $\sum_{i=1}^{h+2} i = \frac{(h+3)\cdot(h+2)}{2}$. As there are 8 faces in an octahedron there are $O(h^2)$ number of vectors generated as a result from an h-subdivided octahedron.

For each normal direction $\overrightarrow{n}$ a set of parallel planes P is constructed. These parallel planes are spaced τ millimetres apart, and span the complete depth of geometry g in the direction of the normal. The set of plane-locations is defined as

$$\left\{r \in \tau\mathbb{Z} \mid \min_{v \in t \in g}\{v \cdot \overrightarrow{n}\} \leq r \leq \max_{v \in t \in g}\{v \cdot \overrightarrow{n}\}\right\} \tag{18}$$

For each scalar r a $\mathsf{HyperPlane}^3$ is constructed by multiplying r by the plane normal $\overrightarrow{n}$.

3.3.2 Basic Partition Search

To find a partitioning a greedy search method is proposed. Input to this method is a geometry g and the number of partition parts n. For this geometry g the plane p from a collection candidate

partition-planes P is found such that the maximum print time per-part according to the heuristic of the partition parts $g^{\top}$ and $g^{\perp}$ is minimized, if $g^{\top}$ were to be partitioned in $n^{\top} = \lceil \frac{n}{2} \rceil$ parts and $g^{\perp}$ were to be partitioned in $n^{\perp} = \lfloor \frac{n}{2} \rfloor$ parts. The print time per part is calculated by dividing the total print time of model g by the number of partition-parts. This estimate will later be improved in section 3.3.2. During the evaluation of all planes the following invariant is maintained: variable p_{best} contains the best plane of all processed candidate planes. The method is recursively executed on the resulting $g^{\top}$ and $g^{\perp}$ geometries with $n^{\top}$ and $n^{\perp}$ number of parts respectively. This is repeated until $n = 1$ in each branch of the algorithm. Pseudo code for this algorithm is given in Algorithm 3.

Algorithm 3: PARTITION SEARCH

Input : Geometry g, the number of available printers n, number of octahedron subdivisions h and parallel planes separation τ in mm
Output: A BSP-tree partitioning geometry g

1 **Function** `PartitionSearch`(g, n) **is**
2   **if** n = *1* **then**
3   **return** BSP-leaf
4   **else**
5     $n^{\top} \leftarrow \lceil \frac{n}{2} \rceil$
6     $n^{\perp} \leftarrow \lfloor \frac{n}{2} \rfloor$
7     $t_{\text{best}} \leftarrow \infty$
8   Initialize p_{best} to a HyperPlane[3]
9   **foreach** $\overrightarrow{n} \in$ `GenNormals`(h) **do**
10   Let P be the set of parallel planes with surface normal $\overrightarrow{n}$ spaced τ mm such that the planes from P span the complete depth of geometry g
11       $V^{\top}$, $V^{\perp} \leftarrow$ `Volumes`(P, g)
12       $A^{\top}$, $A^{\perp} \leftarrow$ `Areas`(P, g)
13   **for** $i = 1$ **to** k **do**
14         $t^{\top} \leftarrow \frac{\texttt{PrintTime}_{\texttt{HARPE}\Omega}(v_i^{\top},\ a_i^{\top},\ 0)}{n^{\top}}$
15         $t^{\perp} \leftarrow \frac{\texttt{PrintTime}_{\texttt{HARPE}\Omega}(v_i^{\perp},\ a_i^{\perp},\ 0)}{n^{\perp}}$
16         $t_{\text{parallel}} \leftarrow \max(t^{\top}, t^{\perp})$
17   **if** $t_{parallel} < t_{best}$ **then**
18           $t_{\text{best}} \leftarrow t_{\text{parallel}}$
19           $p_{\text{best}} \leftarrow p_i$
20   **end**
21   **end**
22   **end**
23     $g^{\top}$, $g^{\perp} \leftarrow$ `Partition`(g, p_{best})
24     $BSP^{\top} \leftarrow$ `PartitionSearch`($g^{\top}$, $n^{\top}$)
25     $BSP^{\perp} \leftarrow$ `PartitionSearch`($g^{\perp}$, $n^{\perp}$)
26   **return** BSP-node constructed from p with left child $BSP^{\top}$ and right child $BSP^{\perp}$
27   **end**
28 **end**

3.3.3 Average Cross-section Area Heuristic

Throughout the algorithm an estimate on the print time *per part* is frequently calculated. This is not as simple as dividing the total print time by the number of parts as each cut increases the total surface area, and consequently the total print time of whole model is increased. By estimating the increase in surface area of all cuts the print times can be more accurately predicted. This is done by looking at the average cross-section area.

If a model is partitioned, the amount of volume in this part turns out to be a great indication on the average cross-section; $\frac{\texttt{AverageCrossSectionArea}(g_{part})}{\texttt{AverageCrossSectionArea}(g)} \approx \frac{\texttt{Volume}(g_{part})}{\texttt{Volume}(g)}$. This is tested empirically on a large data set of models. This set of geometries is the same set used for the results, details of which can be read in section 4. The geometries include some models where one axis is many times larger than the other two axes. The relation between volume and average cross-section area of partitioned geometries is shown in fig. 9. For the heuristic this relation is assumed to be linear.

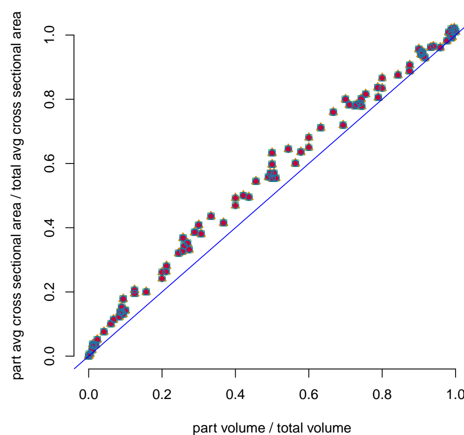


Figure 9: Cross-section area heuristic.

Using the relation between the cross-section and the volume we can estimate the increase in surface resulting from the cuts in a BSP. The area of each cut is assumed to be this average cross-section area. After partitioning each part is then assumed to be half the volume of the original geometry. The average cross-section of these parts is half of the original average cross-section. For each successive layer deeper in the BSP tree the average cross-section area halves. This is illustrated in fig. 10.

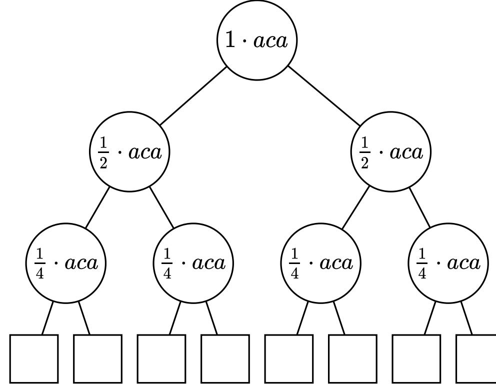


Figure 10: Cross-section area heuristic.

The average sum of the total increase by n cuts is equal to $\log_2(n) \cdot \texttt{AverageCrossSectionArea}(g)$. Section 3.2.4 shows that the print time can be calculated as a function of volume and area. The calculated average increase in surface area can simply be added to the area of g. A more accurate estimate on the parallel print time of a n partitioned model can be calculated using eq. (19).

$$\frac{\texttt{PrintTime}_{\texttt{HARPE}^{\Omega}}(\ \texttt{Volumes}(g),\ \texttt{Area}(g) + \log_2(n) \cdot \texttt{AverageCrossSectionArea}(g)\ )}{n} \tag{19}$$

The print time calculated by the heuristic will generally be higher than the final result of the algorithm. In this heuristic the area of each cut is assumed to be equal to the average cross-section area. As we aim to find the partition that minimizes the maximum print time of any partition part, cuts with a lower cross-section area will be favoured in the algorithm.

The average cross-section area is approximated by sampling the cross-section area geometry using different planes. For a number of normals a collection of parallel planes is created that intersect the geometry. For each of these planes the cross-section area is calculated. The result of the function is the average of all cross-sections. This method is shown in Algorithm 4.

Algorithm 4: Find average cross-section area

Input : Geometry g, number of octahedron subdivisions h and parallel planes separation τ in mm

Output: Average cross-section area in mm^2

1 **Function** AverageCrossSectionArea(g) **is**
2 aca, $c \leftarrow 0$
3 **foreach** $\vec{n} \in$ GenNormals(h) **do**
4 Let P be the set of parallel planes with surface normal $\vec{n}$ spaced τ mm such that the planes from P span the complete depth of geometry g in the normal direction.
5 **foreach** area $\in$ CrossSectionalAreas(P, g) **do**
6 aca += area
7 c += 1
8 **end**
9 **end**
10 **return** $\frac{\text{aca}}{c}$
11 **end**

Using this heuristic the PARTITION SEARCH method is improved. Pseudo code for this improved method is given in Algorithm 5.

Algorithm 5: PARTITION SEARCH with average cross-section area heuristic

```
   Input  : Geometry g, the number of available printers n, number of octahedron
            subdivisions h and parallel planes separation τ in mm
   Output: A BSP-tree partitioning geometry g
 1 Function PartitionSearch_aca(g, n) is
 2     if n = 1 then
 3         return BSP-leaf
 4     else
 5         n^⊤ ← ⌈n/2⌉
 6         n^⊥ ← ⌊n/2⌋
 7         aca ← AverageCrossSectionArea(g)
 8         t_best ← ∞
 9         Initialize p_best to a HyperPlane^3
10         foreach n⃗ ∈ GenNormals(h) do
11             Let P be the set of parallel planes with surface normal n⃗ spaced at τ mm's
                apart such that the planes from P span the complete depth of geometry g
12             V^⊤, V^⊥ ← Volumes(P, g)
13             A^⊤, A^⊥ ← Areas(P, g)
14             for i = 1 to k do
15                 aca^⊤ ← aca · v_i^⊤ / (v_i^⊤ + v_i^⊥)
16                 aca^⊥ ← aca · v_i^⊥ / (v_i^⊤ + v_i^⊥)
17                 t^⊤ ← PrintTime_HARPEΩ(v_i^⊤, a_i^⊤ + log_2(n^⊤) · aca^⊤, 0) / n^⊤
18                 t^⊥ ← PrintTime_HARPEΩ(v_i^⊥, a_i^⊥ + log_2(n^⊥) · aca^⊥, 0) / n^⊥
19                 t_parallel ← max(t^⊤, t^⊥)
20                 if t_parallel < t_best then
21                     t_best ← t_parallel
22                     p_best ← p_i
23                 end
24             end
25         end
26         g^⊤, g^⊥ ← Partition(g, p_best)
27         BSP^⊤ ← PartitionSearch_aca(g^⊤, n^⊤)
28         BSP^⊥ ← PartitionSearch_aca(g^⊥, n^⊥)
29         return BSP-node constructed from p_best, with left child BSP^⊤ and right child
            BSP^⊥
30     end
31 end
```

3.3.4 Support

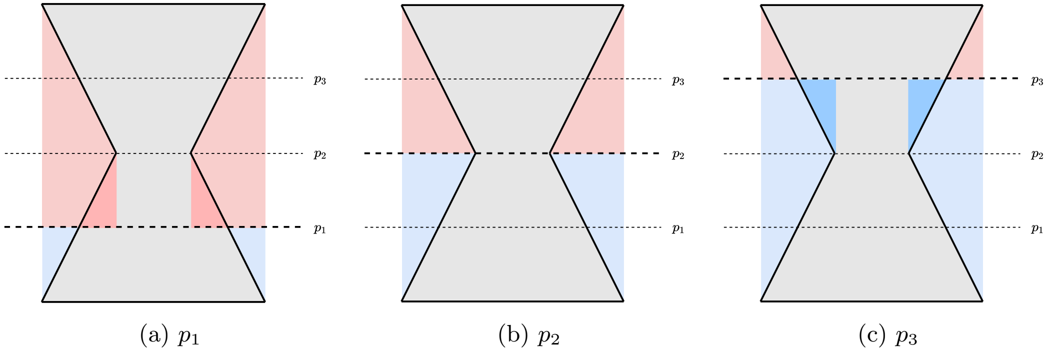
The previous sections did not consider the impact of the support structure on the print time. This section improves the previous two methods by taking into account the print time spent on this feature.

Each partition-plane can serve as a print base. As the print bed of a 3D printer is flat, the part of the model that is printed first and adhered to the bed benefits from being flat. A partition plane proves to be a great print base as the cut results in a flat surface on the partition-part. A partition-part can be printed using any plane at the tree-node from the partition-part's tree-leaf to the BSP-tree's root.

A HyperPlane[3] is used as a definition for a print base. This plane is located such that the complete geometry g is located at the positive side of the plane. As mentioned in section 3.2 the support volume is the sum of irregular triangular prism volumes of all triangles oriented towards the print base.

Each plane p of the candidate cuts P may be used as a print base. If geometry g were to be partitioned by a candidate partition plane p then p may serve as the print base for geometry $g^{\top}$ while $-p$ may serve as the print base for geometry $g^{\perp}$. When evaluating the candidate partition planes P the support volumes for the $g^{\top}$ and $g^{\perp}$ geometries are calculated for each plane $p \in P$ (fig. 11). As we will later see in section 3.4 these support volumes can be efficiently calculated when batched.

This function, $\texttt{SupportVolumes}_{\texttt{geometry}}(P,\ g)$, takes two arguments: a set of parallel planes P and a geometry g. Each plane p_i in P partitions g in geometry-parts $g_i^{\top}$, $g_i^{\perp}$. The *support volumes geometry* operation returns the support volumes for the $g_i^{\top}$ and $g_i^{\perp}$ geometries. The print base used when calculating the support volume is p_i for geometry $g_i^{\top}$, and $-p_i$ for geometry $g_i^{\perp}$.

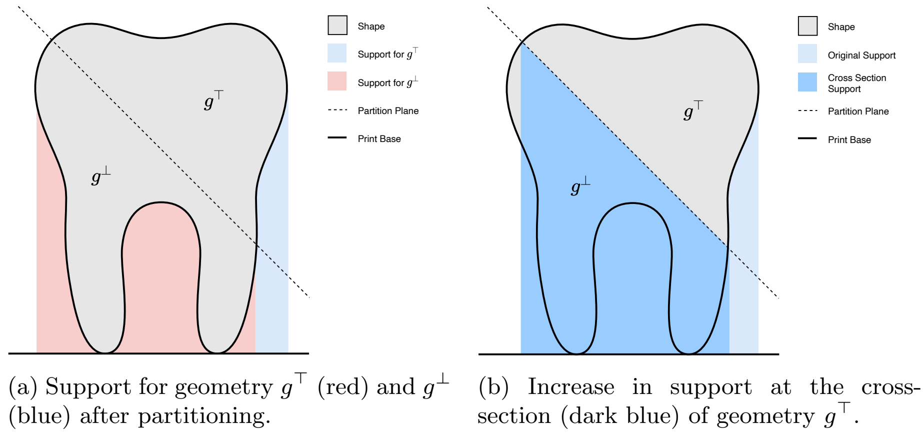


(a) p_1 (b) p_2 (c) p_3

Figure 11: Support structure for $g^{\top}$ (red) and $g^{\perp}$ (blue) when partitioning geometry g using plane p_1 (fig. 11a), p_2 (fig. 11b) and p_3 (fig. 11c).

In addition to the print base generated from the candidate partition planes all planes created from previous cut may serve as a print base. The print direction for these print bases are generally oriented differently from the candidate partition planes.

When a candidate partition plane is evaluated it may choose the support structure of one of the print bases created earlier in the algorithm. The partition plane will in addition to the geometry also partition the support structure (fig. 12a). Unless partition plane p is orthogonal to the print base, p will expose a surface at either part where additional support is needed. This additional support is located at the cross-section between partition plane p and geometry g (fig. 12b).



(a) Support for geometry $g^{\top}$ (red) and $g^{\perp}$ (blue) after partitioning.

(b) Increase in support at the cross-section (dark blue) of geometry $g^{\top}$.

Figure 12: Partitioning of the support structure.

The Partition Search method with support (Algorithm 6) is similar to Algorithm 5, but with some additions. The function takes one additional argument; the set of print-bases P^{bases}. This argument is initially empty and will in later steps contain a set of candidate print bases. Each time a geometry g is partitioned the partition plane p used to cut g also serves as candidate partition plane. For partition parts $g^{\top}$ and $g^{\perp}$ print bases p and $-p$ are added respectively.

For each of the candidate cuts $p \in P$ the support volume is calculated if the plane p were to be used as print base (line 15). Then for each of the provided print bases (line 16) the support is partitioned by the candidate planes P (line 17). If the support volume when using one of these partition planes improves the current support volume, the current support volume is updated (line 19, line 20). This minimal support volume of all print bases is used when calculating the print time (line 26, line 27).

Algorithm 6: Partition Search with support

Input : Geometry g, the number of available printers n, a set of candidate print bases, number of octahedron subdivisions h and parallel planes separation τ in mm

Output: A BSP-tree where the print direction of each part is stored in the leaves

1 **Function** PartitionSearch$_{\text{support}}$(g, n, $P^{base} = \varnothing$) **is**
2   **if** n = 1 **then**
3   Let p_{best} be the print base from P^{base} that minimizes the print time
4   **return** BSP-leaf containing p_{best}
5   **else**
6     $n^{\top} \leftarrow \lceil \frac{n}{2} \rceil$
7     $n^{\perp} \leftarrow \lfloor \frac{n}{2} \rfloor$
8   aca $\leftarrow$ AverageCrossSectionArea(g)
9     $t_{\text{best}} \leftarrow \infty$
10   Initialize p_{best} to a HyperPlane[3]
11   **foreach** $\overrightarrow{n} \in$ GenNormals(h) **do**
12   Let P be the set of parallel planes with surface normal $\overrightarrow{n}$ spaced at τ mm's apart such that the planes from P span the complete depth of geometry g
13       $V^{\top}$, $V^{\perp} \leftarrow$ Volumes(P, g)
14       $A^{\top}$, $A^{\perp} \leftarrow$ Areas(P, g)
15   supportVolumes$^{\top}$, supportVolumes$^{\perp} \leftarrow$ SupportVolumes$_{\text{geometry}}$(P, g)
16   **foreach** $p^{base} \in P^{base}$ **do**
17   supportVolumesSupport $\leftarrow$ SupportVolumes$_{\text{support}}$(P, p^{base})
18   **for** $i = 1$ **to** k **do**
19           $\text{supportVolumes}_i^{\top} \leftarrow \min(\text{supportVolumes}_i^{\top}, \text{supportVolumesSupport}_i^{\top})$
20           $\text{supportVolumes}_i^{\perp} \leftarrow \min(\text{supportVolumes}_i^{\perp}, \text{supportVolumesSupport}_i^{\perp})$
21   **end**
22   **end**
23   **for** $i = 1$ **to** k **do**
24         $\text{aca}^{\top} \leftarrow \text{aca} \cdot \frac{v_i^{\top}}{v_i^{\top} + v_i^{\perp}}$
25         $\text{aca}^{\perp} \leftarrow \text{aca} \cdot \frac{v_i^{\perp}}{v_i^{\top} + v_i^{\perp}}$
26         $t^{\top} \leftarrow \frac{\text{PrintTime}_{\text{HARPE}}\Omega(v_i^{\top},\ a_i^{\top} + \log_2(n^{\top}) \cdot \text{aca}^{\top}, \text{supportVolumes}_i^{\top})}{n^{\top}}$
27         $t^{\perp} \leftarrow \frac{\text{PrintTime}_{\text{HARPE}}\Omega(v_i^{\perp},\ a_i^{\perp} + \log_2(n^{\perp}) \cdot \text{aca}^{\perp}, \text{supportVolumes}_i^{\perp})}{n^{\perp}}$
28         $t_{\text{parallel}} \leftarrow \max(t^{\top}, t^{\perp})$
29   **if** $t_{parallel} < t_{best}$ **then**
30           $t_{\text{best}} \leftarrow t_{\text{parallel}}$
31           $p_{\text{best}} \leftarrow p_i$
32   **end**
33   **end**
34   **end**
35     $g^{\top}$, $g^{\perp} \leftarrow$ Partition(g, p_{best})
36     $BSP^{\top} \leftarrow$ PartitionSearch$_{\text{support}}$($g^{\top}$, $n^{\top}$, printBases $\cup \{p_{best}\}$)
37     $BSP^{\perp} \leftarrow$ PartitionSearch$_{\text{support}}$($g^{\perp}$, $n^{\perp}$, printBases $\cup \{-p_{best}\}$)
38   **return** BSP-node constructed from p_{best}, with left child $BSP^{\top}$ and right child $BSP^{\perp}$
39   **end**
40 **end**

3.4 Batched Calculations

During the PARTITION SEARCH method certain properties are calculated on a number of candidate cuts. The number of candidate cuts influence the quality of the partitioning; a higher number of candidate cuts leads to a better result as more options are evaluated. As these candidate partition-planes are parallel it is possible to group a batch of planes to efficiently calculate these properties.

The candidate cuts P consist of k parallel planes; $P = p_1, p_2, \ldots, p_k$. The distance between each plane in τ mm. Each plane p_i serves as a candidate partition plane that, if chosen, would partition the geometry into parts $g_i^{\top}$ and $g_i^{\perp}$. Before picking a partition plane the best one is chosen based on certain properties such as each geometry-parts' volume, area etc. The batched calculations allow to calculate these features of each geometry-part $g_i^{\top}$ and $g_i^{\perp}$ *without* partitioning.

The operations for which the batched performance can be improved are the *cross-sectional areas* (section 3.4.1), *surface areas* (section 3.4.2), *volumes* (section 3.4.3), *support volumes' geometry* (section 3.4.4) *support volumes' support* (section 3.4.5) calculations. The cross-sectional, and surface area calculations are arguably trivial, they do however serve as nice introduction for the volume calculations.

When naively implementing these operations, by having an outer loop through all parallel planes and an inner loop that calculates for each plane the intersection with each triangle of the input geometry, the time complexity accumulates to $O(mk)$. Here m is the number of triangles in geometry g and k is the number of parallel planes. The main idea behind the batching operations is reversing the natural order of these two loops. Reversing the order of the loops allows to more efficiently determine what planes have intersections with what triangles, eliminating a lot of intersection checks. The complexity when batching these operations reduces the complexity from $O(mk)$ to $O(m + k + c)$ where c is the number of plane-triangle intersections. The number of intersections is bounded by mk intersections, but for most practical models c is a lot lower[2].

Section 3.2.4 shows that the print time of a model-part can be estimated as a function of the model's volume and area. Calculating the print time becomes very efficient as a result from the batched operations as both the volume and surface area can be calculated.

3.4.1 Cross-sectional Area

The function `CrossSectionalAreas` calculates the area of the cross-section between geometry g and each plane p_i in a set of parallel planes P.

Naive Approach Calculating the cross-sectional area between a *single* plane v and geometry g is fairly straight forward. First for each triangle $t \in g$ that is intersecting g the planar intersection is calculated. This is the intersection in the plane's two-dimensional planar space. The result of this intersection is a two-dimensional line segment. Sequencing all these line segments results in a polygon S.

The cross-sectional area can then be calculated by using the *Sum of Signed Areas*[2]. Let S be a polygon containing l points in clockwise order. For each two successive points u and v there is a line segment s. The signed area for a line segment is a parallelogram. Endpoints u and v are connected points directly below u and v at $y = 0$ (fig. 13). The area for this line-segment is

[2]When the parallel planes' separation τ is bigger than the largest diagonal distance of a triangle t, each triangle can at most intersect a single plane, making $c \leq m$ resulting in a time complexity of $O(m + k)$ for the relevant operations.

described in eq. (20). Note that the area is positive if the line segment is directed from left to right and negative if the segment is directed from right to left.

$$\texttt{SignedArea}(s) = (s.v_x - s.u_x)\frac{s.u_y + s.v_y}{2} \tag{20}$$

The total area of polygon S is the sum of these signed areas.

$$\texttt{Area}(S) = \sum_{s \in S} \texttt{SignedArea}(s) \tag{21}$$

An intuition as to why this produces the correct area is illustrated in fig. 14. Each line segment going from left to right casts a positive area shown in blue in fig. 14. These areas cover a super set of the area of polygon S. Exactly those areas that are in this super set but not in the original polygons are covered by the negative areas cast by the segments going from right to left shown in red in fig. 14.

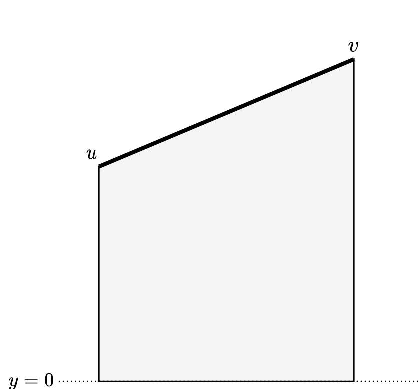


Figure 13: Area for a line segment.

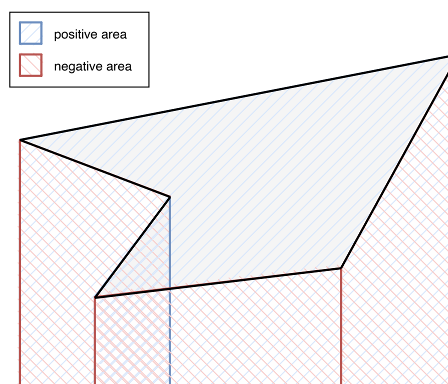


Figure 14: Polygon shown in grey, with for each line segment the signed area highlighted.

For Partition Search we are interested in the cross-sectional area for a set of parallel planes P. When naively implementing this variant by repeating the algorithm described above for all k planes, the time complexity results in $O(mk)$. This time complexity can be improved by batching the planes.

Batched Approach For each triangle t in g the lower most and upper most plane that is intersecting t is found. This can be done in constant time as the planes are sorted and spaced at equal distances. As these planes are defined in the algorithm we have control over the spacing and ordering of the planes.

For each of the intersecting planes the signed areas are calculated similarly to the single plane cross-sectional area calculation. As addition is commutative, re-ordering the summation does not change its outcome. It is thus not required to traverse the line segments in order, but any ordering is possible. Pseudo code for this algorithm is given in Algorithm 7.

Algorithm 7: Batched cross-sectional areas

Input : Geometry g consisting of m triangles $t_1, t_2, \ldots, t_m$, k parallel planes $P = p_1, p_2, \ldots, p_k$

Output: List areas $A = a_1, a_2, \ldots, a_k$ where each area a_i is the cross-section area between plane p_i and geometry g

```
1  Function CrossSectionalAreas(g, P) is
2  |  Initialize A to a_i ← 0 for all i = 1, 2, ..., k
3  |  foreach t ∈ g do
4  |  |  Let l and u correspond to the lower and upper most plane index that is
   |  |   intersecting t
5  |  |  for i = l to u do
6  |  |  |  s ← PlanarIntersect(t, p_i)
7  |  |  |  a_i += SignedArea(s)
8  |  |  end
9  |  end
10 |  return A
11 end
```

3.4.2 Surface Areas

Another operation used frequently during Partition Search is to find the areas of all geometries $g_i^\top$ and $g_i^\perp$ when partitioning g using candidate partition planes $P = p_1, p_2, \ldots, p_k$.

This is done by using *bins*; spaces between planes (see fig. 15). The batched algorithm involves calculating the surface area within each bin. As there are k planes there are $k+1$ bins. For each triangle $t \in g$ the contribution of t for each bin b is calculated and added to b. Once the surface area within bins b_j, $j = 1, 2, \ldots, k+1$ is known, the surface area of $g_i^\top$ (if g were to be partitioned by plane p_i) is equal to the *accumulation* of bins from $i+1$ to k; $\sum_{j=i+1}^{k+1} b_j$. The surface area of $g_i^\perp$ is equal to $\sum_{j=1}^{i} b_j$. In Algorithm 8 pseudo code for calculating the surface area is shown.

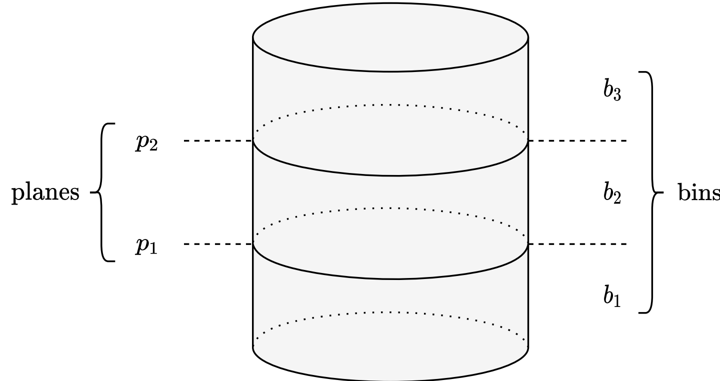


Figure 15: Showing bins for $k = 2$ planes intersecting geometry g.

Algorithm 8: Batched Surface Areas

Input : Geometry g consisting of m triangles $t_1, t_2, \ldots, t_m$, k parallel planes $P = p_1, p_2, \ldots, p_k$

Output: List of surface areas for geometry parts $g_i^\top$ and $g_i^\perp$ if geometry g were to be partitioned by all planes p_i

```
1  Function Areas(g, P) is
2      Initialize B where b_i ← 0 for all i = 1, 2, ..., k + 1
3      foreach t ∈ g do
4          Let l and u be the bin indices corresponding to the lower and upper most vertices
            of t
5          for i = l to u do
6              Let a be the area of t between planes p_{i−1} and p_i
7              b_i += a
8          end
9      end
10     Let A^⊤ be the accumulation a_i^⊤ ← Σ_{j=i+1}^{k+1} b_j for all i = 1, 2, ..., k
11     Let A^⊥ be the accumulation a_i^⊥ ← Σ_{j=1}^{i} b_j for all i = 1, 2, ..., k
12     return A^⊤, A^⊥
13 end
```

When g is not just a surface mesh but a solid mesh, the cross-sectional area of the plane cut contributes to the total surface area. For these kinds of geometries the cross-sectional areas of each plane need to be added to the top and bottom surface areas of each cut. This is demonstrated in Algorithm 9.

Algorithm 9: Batched surface areas for solid geometries

Input : Geometry g consisting of m triangles $t_1, t_2, \ldots, t_m$, k parallel planes $P = p_1, p_2, \ldots, p_k$

Output: List of surface areas for geometry parts $g_i^\top$ and $g_i^\perp$ if geometry g were to be partitioned by all planes p_i

```
1  Function AreasSolid(g, P) is
2      A^⊤, A^⊥ ← Areas(g, P)
3      A^cross-section ← CrossSectionalAreas(g, P)
4      for i = 1 to k do
5          a_i^⊤ += a_i^cross-section
6          a_i^⊥ += a_i^cross-section
7      end
8      return A^⊤, A^⊥
9  end
```

3.4.3 Volume

Another operation that can be improved by batching is the volume operation. Here the volumes for $g_i^\top$ and $g_i^\perp$ are calculated for each plane p_i.

Sum of Signed Volumes The formula for calculating the area can be generalized to work in higher dimensions[2]. For three-dimensional geometries consisting of triangles the volume can be calculated using the *sum of signed volumes* of each triangle t. This is done by connecting

each vertex v of triangle t to location $z = 0$ directly below v. The shape that this creates is an *irregular triangular prism* (fig. 16).

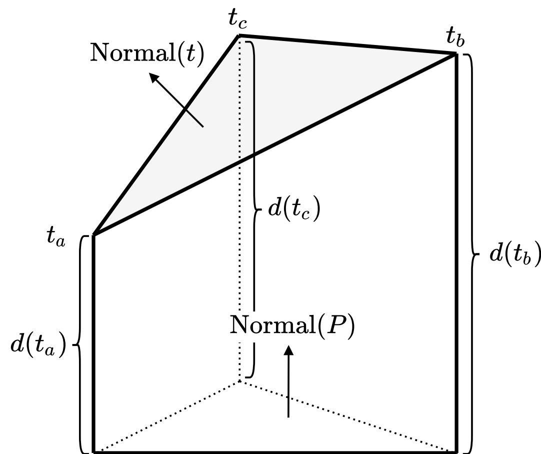


Figure 16: Triangle t with accompanied irregular prism.

To calculate the volume of such an irregular prism we need to know the distance between each triangle-vertex and the base. For reasons that will become apparent later this base is angled, this angle is the normal of the parallel planes. The distances of these vertices are calculated using the formula

$$d(v) = \texttt{Normal(}P\texttt{)} \boldsymbol{\cdot} v \tag{22}$$

where $\texttt{Normal(}P\texttt{)}$ is the normal of the parallel planes and v is one of the vertices of triangle t. By multiplying the area of t by the projection of the plane-normal $\texttt{Normal(}P\texttt{)}$ on the triangle-normal $\texttt{Normal(}t\texttt{)}$ we get the signed projected area of t on a plane. This is signed because the area becomes negative when the two normals point in opposite direction. The volume of this *irregular triangular prism* is

$$\texttt{SignedVolume}(t) = \frac{\texttt{Area(}t\texttt{)} \cdot \texttt{Normal(}t\texttt{)} \boldsymbol{\cdot} \texttt{Normal(}P\texttt{)} \cdot (d(t_a) + d(t_b) + d(t_c))}{3} \tag{23}$$

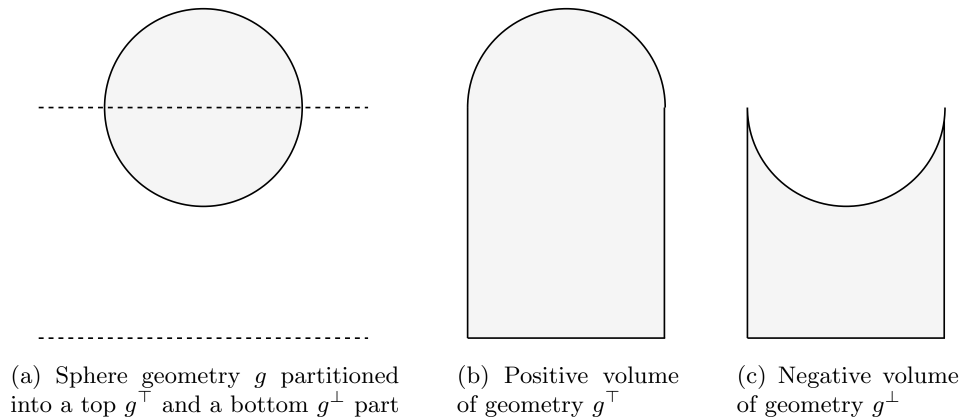
The total volume of g is calculated as the sum

$$\texttt{Volume(}g\texttt{)} = \sum_{t \in g} \texttt{SignedVolume}(t) \tag{24}$$

Bin Volumes Similar to before the space is split into $k+1$ bins. The approach works similarly as the previous approach by evaluating each triangle t from geometry g. For each bin space triangle t intersects, the signed volume contained within this bin is calculated. This volume is added to the relevant bin.

Calculating the volumes like this produces the incorrect results as only the volume for part of the model is calculated; only the sum of signed volumes of *all* triangles of a manifold geometry results in the correct volume of the geometry. The intersection between all surface faces and a bin space does not result in a manifold mesh.

In fig. 17a a geometry g is partitioned using plane p into a top $g^{\top}$ geometry and a bottom $g^{\perp}$ geometry. As all triangles in $g^{\top}$ point in the same direction as the plane normal the signed volume for $g^{\top}$ can be visualised as the shape shown in fig. 17b. Similarly the volume for $g^{\perp}$ is incorrect. As all normals from $g^{\perp}$ point in opposite direction of the plane normal, the signed volume of $g^{\perp}$ is negative and can be visualized as shown in fig. 17c.



(a) Sphere geometry g partitioned into a top $g^{\top}$ and a bottom $g^{\perp}$ part

(b) Positive volume of geometry $g^{\top}$

(c) Negative volume of geometry $g^{\perp}$

Figure 17: Signed volumes when partitioning a sphere geometry.

For both geometries $g^{\top}$, and $g^{\perp}$ the signed volume is incorrect as the geometry of the cross-section is missing. For correct volume calculations the signed volume of these missing cross-sections needs to be added. By making the base the same normal as the parallel plane this volume can easily calculated using the cross-section area calculations discussed previously in section 3.4.1. The volume of the missing cross-section is equal to the cross-section area of partition plane p and geometry g multiplied by the distance of p from the base. As the normal of the cross-section of $g^{\top}$ is pointing in opposite direction of the plane the signed volume for $g^{\top}$ is negative, and as the normal of the cross-section of $g^{\perp}$ is pointing in the same direction of the normal of the plane the signed volume for $g^{\perp}$ is positive.

Pseudo code for calculating the volumes is shown in Algorithm 10.

Algorithm 10: Batched Volumes

Input : Geometry g consisting of m triangles $t_1, t_2, \ldots, t_m$ and k parallel planes $P = p_1, p_2, \ldots, p_k$ spaced τ millimetres apart

Output: List of volumes for geometry parts $g_i^\top$ and $g_i^\perp$ if geometry g were to be partitioned by all planes p_i

```
1  Function Volumes(g, P) is
2      Initialize B to b_i ← 0 for all i = 1, 2, ..., k + 1
3      foreach t ∈ g do
4          Let l, u be the bin indices corresponding to the lower and upper most vertices of t
5          for i = l to u do
6              Let v be the signed volume of the part of t between planes p_{i−1} and p_i
7              b_i += v
8          end
9      end
10     Let V^⊤ be the accumulation v_i^⊤ ← Σ_{j=i+1}^{k+1} b_j for all i = 1, 2, ..., k
11     Let V^⊥ be the accumulation v_i^⊥ ← Σ_{j=1}^{i} b_j for all i = 1, 2, ..., k
12     A^{cross-section} ← CrossSectionalAreas(g, S)
13     for i = 1 to k do
14         v_i^⊤ −= d(p_i) · a_i^{cross-section}
15         v_i^⊥ += d(p_i) · a_i^{cross-section}
16     end
17     return V^⊤, V^⊥
18 end
```

3.4.4 Support Volumes' Geometry

The *support volumes' geometry* operation calculates the support volumes when partitioning a geometry for each plane out of a set of parallel planes P. The support volume for part $g_i^\top$ uses plane p_i for the print base and part $g_i^\perp$ uses plane $-p_i$ for the print base.

During the operation each triangle $t \in g$ is assigned a print base. Only if a triangle is oriented such that `Normal(`t`)` is in opposite direction to `Normal(`P`)` then support is needed when using `Normal(`P`)` as print direction. Alternatively when using print direction $-$`Normal(`P`)` then all triangles oriented such that `Normal(`t`)` is the same direction as `Normal(`P`)` need support.

When the support volume is calculated all triangles $t \in g$ are traversed. For each triangle t the print direction for which support is needed is determined. First the case where the support is needed in the print direction is considered. Then the support volumes that are within bins that intersect t are calculated (fig. 18).

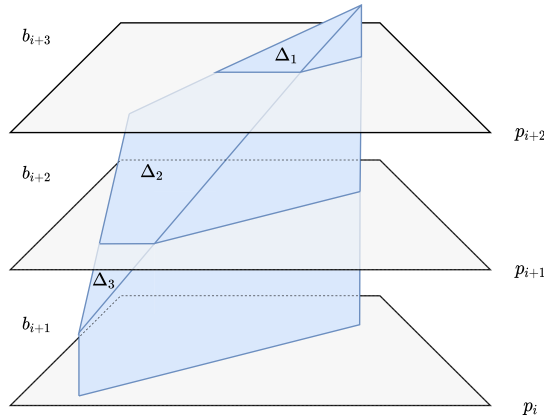
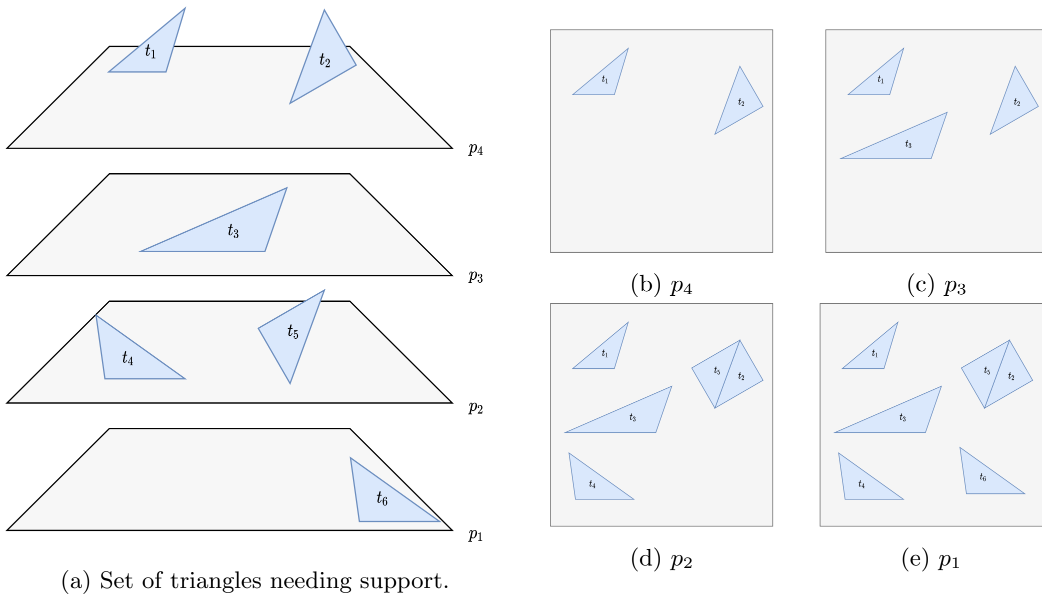


Figure 18: Support volume needed for a single triangle t, partitioned using parallel planes P.

The support volume would be correctly calculated when calculating the support for t when continuing until plane p_1. However, this would be too costly. Instead the projected area of t on P is calculated and added to a *cumulative area* bin. This is shown in fig. 19. Figure 19a shows a set of triangles needing support. Interspersed with these triangles is a set of parallel planes P placing each triangle in a bin. Figures 19b, 19c, 19d and 19e depict the cumulative support area for planes p_4, p_3, p_2 and p_1 respectively. As the support is generated from a triangle all the way down to the print base, the accumulated support is calculated by traversing from the upper most bin down to the lower most bin. Let a_i be the addition to this cumulative support area for plane p_i, then the cumulative support volume for plane p_i can be calculated as $c_i = \sum_{j=i}^{k} a_j$. The area of support within a bin b_i is the cumulative support area for the plane directly above b_i multiplied by the separation of the planes τ; $b_i = c_j \cdot \tau$.



(a) Set of triangles needing support.

(b) p_4

(c) p_3

(d) p_2

(e) p_1

Figure 19: Cumulative support area for a set of triangles T for a selection of parallel planes P.

In Algorithm 11 pseudo code for calculating the support volumes for a geometry is given.

Algorithm 11: Batched Support Volume Geometry

```
   Input  : Geometry g consisting of m triangles t_1, t_2, ..., t_m and k parallel planes
            P = p_1, p_2, ..., p_k spaced τ millimetres apart
   Output: List of top and bottom support volumes for models g^⊤ and g^⊥ for each plane
            p_i if plane p_i were to be used as print base
 1 Function SupportVolumes_geometry(g, P) is
 2     Initialize B^⊤ and B^⊥ to b_i^⊤, b_i^⊥ ← 0 for all i = 1, 2, ..., k + 1
 3     Initialize A^⊤ and A^⊥ to a_i^⊤, a_i^⊥ ← 0 for all i = 1, 2, ..., k
 4     foreach t ∈ g do
 5         Let l, u be the bin indices corresponding to the lower and upper most vertices of t
 6         for i = l to u do
 7             Let v be the support volume needed for triangle-part of t that is between
                 planes p_{i−1} and p_i
 8             Increase either b_i^⊤ or b_i^⊥ by v depending for what direction support for t is
                 required
 9         end
10         Let s be the projected area of t on P
11         Increase either a_i^⊤ or a_i^⊥ by s depending for what direction support for t is
             required
12     end
13     Initialize V^⊤ to v_i^⊤ ← 0 for all i = 1, 2, ..., k
14     v_k^⊤ ← b_{k+1}
15     cumulativeArea ← a_k
16     for i = k − 1 to 0 do
17         v_i^⊤ ← v_{i+1}^⊤ + b_{i+1} + cumulativeArea · τ
18         cumulativeArea += a_i^⊤
19     end
20     Initialize V^⊥ to v_i^⊥ ← 0 for all i = 1, 2, ..., k
21     v_1^⊥ ← b_1
22     cumulativeArea ← a_1
23     for i = 2 to k do
24         v_i^⊥ ← v_{i−1}^⊥ + b_i + cumulativeArea · τ
25         cumulativeArea += a_i^⊥
26     end
27     return V^⊤, V^⊥
28 end
```

3.4.5 Support Volumes' Support

When a geometry part is partitioned using a plane p the geometry part can still be printed using previously created support structures. However, when partitioning a geometry into $g^{\perp}$ and $g^{\top}$ then $g^{\perp}$ does not need the support required for $g^{\top}$ and $g^{\top}$ does not need the support required for $g^{\perp}$. The support structure itself needs to be partitioned as well. The *support volumes' support* operation calculates the support volumes when partitioning a support structure for each plane out of a set of parallel planes P.

Algorithm 12: Batched Support Volume Support

Input : Geometry g consisting of m triangles $t_1, t_2, \ldots, t_m$, the print base p^{base} and k parallel planes $P = p_1, p_2, \ldots, p_k$

Output: List of support volumes when using plane p_i as a print base for geometry parts $g_i^\top$ and $g_i^\perp$ if geometry g were to be partitioned by all planes p_i

```
1  Function SupportVolumes_support(g, p^base, P) is
2      Initialize B to b_i ← 0 for all i = 1, 2, ..., k + 1
3      foreach t ∈ g do
4          if triangle t requires support when using p^base then
5              Let l, u be the bin indices corresponding to the lower and upper most vertices
                 of t
6              for i = l to u do
7                  Let v be the volume corresponding the support volume of the polygon
                     between planes p_{i-1} and p_i
8                  b_i += v
9              end
10     end
11     Let V^⊤ be the accumulation v_i^⊤ ← Σ_{j=i+1}^{k+1} b_j for all i = 1, 2, ..., k
12     Let V^⊥ be the accumulation v_i^⊥ ← Σ_{j=1}^{i} b_j for all i = 1, 2, ..., k
13     return V^⊤, V^⊥
14 end
```

3.5 Local Search Procedure

After an initial BSP has been found by the Partition Search method the BSP is refined using a local search procedure. During the search method the partition-planes are immutable; once a cut has been made the cut does not change. Once the complete BSP is constructed it might become apparent that earlier cuts are suboptimal. The Local Search procedure works by iteratively moving each plane in the BSP in the direction of largest average print time. Each iteration results in an improved parallel print time.

Let us first consider the case where a geometry is partitioned using a single plane p. If the resulting geometries differ in print time then a better solution can be found by moving p towards the geometry-part with longer print time. It is easy to see that for the simple case where a geometry is split using a single plane the optimal parallel print time partitions the geometry in two parts with equal print time.

During each iteration the BSP-tree is recursively traversed. The result of each recursion step is the adjusted BSP, the total print time t of all partition-parts and the number of parts g. If a tree-leaf is encountered during the traversal this is easy; the sub-tree contains a single partition part, and the total print time is the print time of said partition part. If alternatively a tree-node is encountered, first the total print time $(t^\top, t^\perp)$ and number of parts $(n^\top, n^\perp)$ is calculated using the recursive method. Then the average time difference per part between the left and right sub-tree δ_t is defined as $\frac{t^\top}{n^\top} - \frac{t^\perp}{n^\perp}$. Plane p' is constructed from p by moving p in the direction of p's normal vector by $\alpha \cdot \delta_t$. As δ_t is measured in seconds and the position of p is measured in millimetres, δ_t is multiplied by a weight α to account for the difference in both domains. The method for adjusting a BSP is shown in alg. 13.

Algorithm 13: Adjust BSP

```
   Input  : A BSP partitioning geometry g and a weight α
   Output: Updated BSP, total print time t and the number of parts n
 1 Function AdjustBSP(BSP, g) is
 2   if IsLeaf(BSP) then
 3     t ← PrintTime(g)
 4     n ← 1
 5     return BSP, t, n
 6   else if IsNode(BSP) then
 7     Let p be the hyper plane at the BSP-node and let BSP^⊤ be its left child and
         BSP^⊥ its right child
 8     Let g^⊤ and g^⊥ be the resulting geometries from g partitioned by p
 9     BSP^⊤, t^⊤, n^⊤ ← AdjustBSP(BSP^⊤, g^⊤)
10     BSP^⊥, t^⊥, n^⊥ ← AdjustBSP(BSP^⊥, g^⊥)
11     δ_t ← t^⊤/n^⊤ − t^⊥/n^⊥
12     p′ ← p
13     p′.d += α · δ_t
14     BSP′ ← tree-node from p′ with left child BSP^⊤ and right child BSP^⊥
15     return BSP′, t^⊤ + t^⊥, n^⊤ + n^⊥
16   end
17 end
```

The adjustment procedure described above is repeated x times, or until the new BSP ceases to improve the old BSP. A typical value for x is 20. Pseudo code for the Local Search procedure is given in Algorithm 14.

Algorithm 14: Local Search

```
   Input  : An initial BSP partitioning geometry g and the number of iterations x
   Output: Updated BSP
 1 Function LocalSearch(BSP, g, x) is
 2   repeat x times
 3     BSP′ ← AdjustBSP(BSP, g)
 4     if max_{g′∈BSP′} {PrintTime_HARPE( g′)} > max_{g″∈BSP} {PrintTime_HARPE( g″)} then
 5       return BSP
 6     end
 7     BSP ← BSP′
 8   end
 9   return BSP
10 end
```

Each iteration yields diminishing results in the parallel print time. This is depicted in fig. 20 for the armadillo model partitioned into $n = 6$ parts. With a higher number of iterations the print times of all parts reach an equilibrium where time spent printing each model is approximately equal. Figure 21 depicts the partitioning for the same model after $x = 0, 1, 2$ iterations. In section 4.4 we will experimentally show the parallel print time improvement of the local search procedure applied to Partition Search partitioned models.

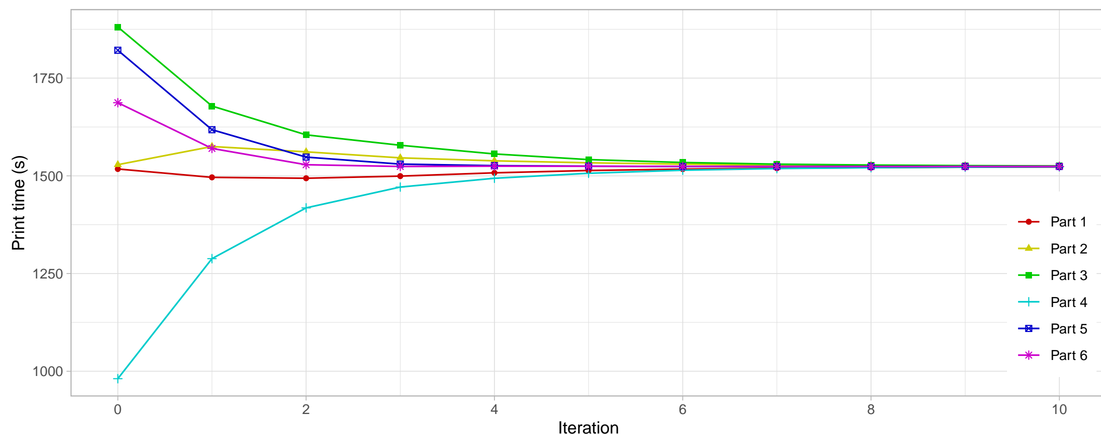
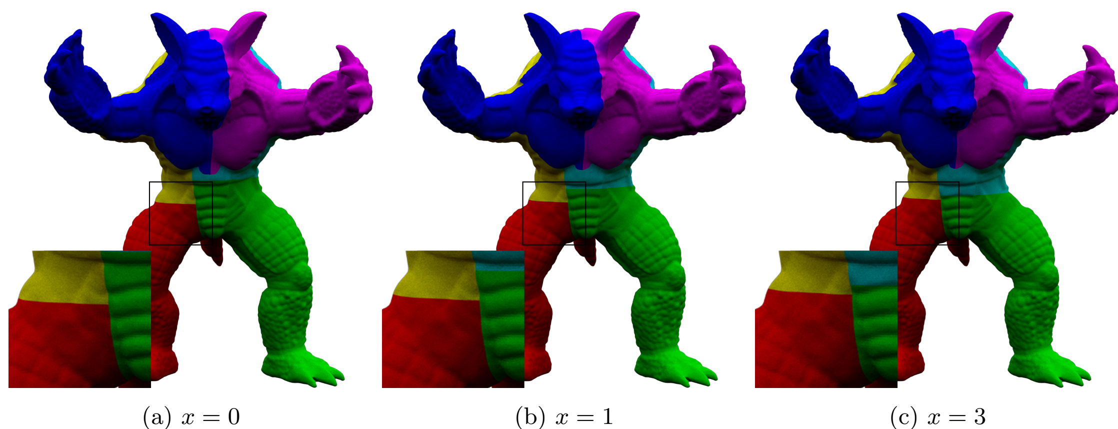


Figure 20: The print time of each of the six model-parts of the armadillo model after local search optimizations of $x = 0, \ldots, 10$ iterations.

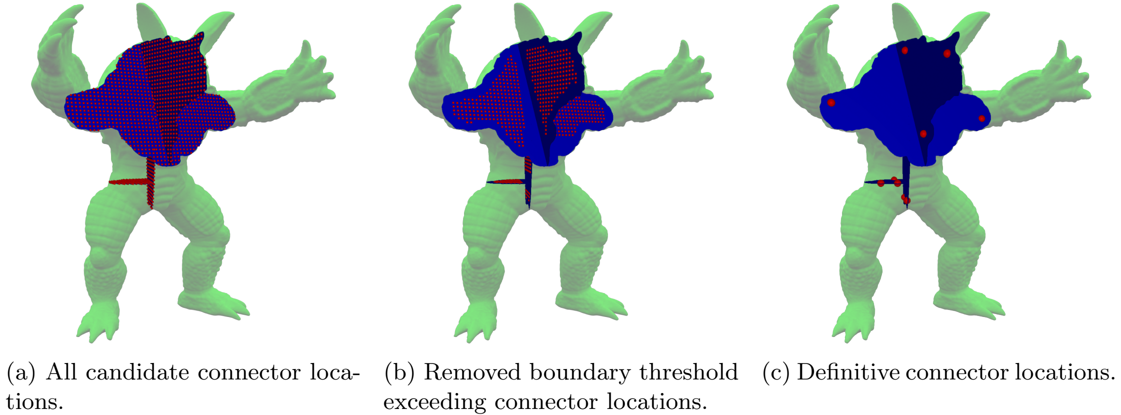


(a) $x = 0$ (b) $x = 1$ (c) $x = 3$

Figure 21: Visualisation of BSP partitions of the armadillo model after local search optimizations of $x = 0, 1, 3$ iterations.

3.6 Connectors

On the cross-section between cuts, connectors are added for an easier assembly process. These connectors are added in the following steps. First a set of candidate connector-locations is found on the cross-sections. These candidate connector locations are located on the surface of each plane in the BSP-tree (fig. 22a). Any connector that is too close to the boundary is removed (fig. 22b). From this set of candidate connector locations the definitive connector locations are found and a connector is added to the geometry part (fig. 22c). In order to preserve a flat print base two female connectors are added to each side of the cross-section. The parts can then be assembled using a pre-manufactured connector-piece.



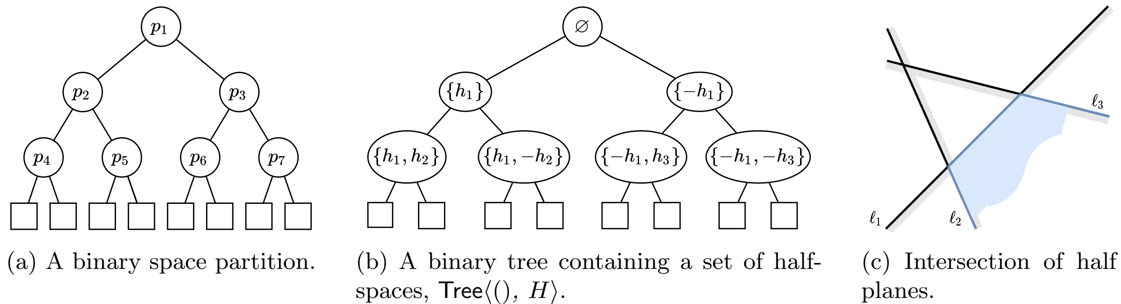
(a) All candidate connector locations.
(b) Removed boundary threshold exceeding connector locations.
(c) Definitive connector locations.

Figure 22: Steps for finding the connector locations on the partition-planes for the armadillo model (green) with partition-planes (blue) and (candidate) connector locations (red).

Let V be the set of candidate connector locations on the partition-planes. The points $v \in V$ that are too close to the boundary, either the to the geometry itself or to one of the partition-planes, are removed. The library we used for our representation of geometries provides methods for such point-distance operations. After some investigation this method is implemented using a quad-tree acceleration structure. This quad-tree is traversed by exploring the nodes closest to the target location first. During this traversal the closest face found thus far is stored. When exploring a quad-tree node, if the closest point of the node is located further from the target location compared to the closest face found thus far, then there is no need to explore the node.

The region of a leaf in a BSP-tree can be expressed as the intersection of half-spaces of all nodes in the path from the leaf to the root of the tree. As such we don't want to remove the points that are in range of the infinite plane p, but instead we only want to remove the points that are in range of p within the region of the BSP leaf. For each BSP-node, the geometry of its partition-plane is created in each BSP-node's planar space and then transformed to a geometry. The half-spaces of all parents for a partition-plane p are intersected with p. The intersecting line is transformed to the planar space of p, resulting in a set of half-planes (fig. 23b). The intersection of these half-planes results in a convex polygon (fig. 23c) that is converted to a geometry, and used for the boundary queries.

The intersection of these half-spaces is calculated using the approach described in [3]. In the approach the set of half-planes is split into two parts, and for each part the intersection is recursively calculated. These two convex polygons are intersected to form the resulting polygon.



(a) A binary space partition.
(b) A binary tree containing a set of half-spaces, $\mathsf{Tree}\langle(),\ H\rangle$.
(c) Intersection of half planes.

Figure 23: Overview for creating the partition-plane geometries.

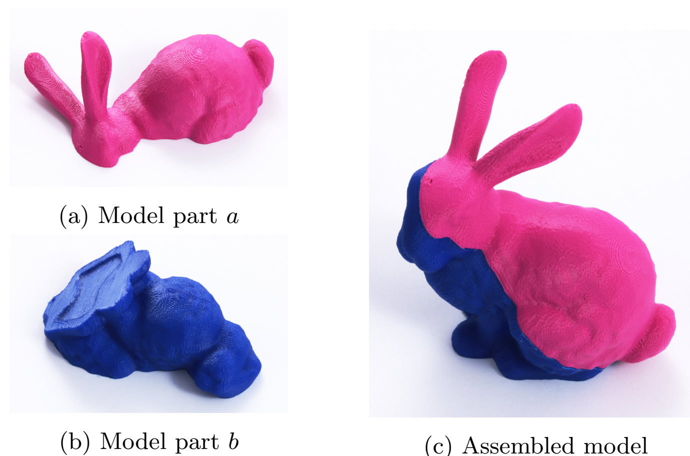
From this set of candidate connector locations V the set of definitive connector locations is found. The definitive connector locations are the locations that are located on the boundary of the smallest enclosing disc of V. To find the smallest enclosing disc the method as described in [4] is used. In this approach the smallest enclosing disc is found in expected linear time using *randomized incremental construction.* The input points are randomized and two points are selected to form an initial disc D. All points $v \in V$ are traversed until a point v_a is encountered for which $v_a \notin D$. Let V_a be the points processed thus far. We know that point v_a is located on the boundary of the smallest enclosing disc of points V_a. A process similar as before is started; the points from V_a are randomized and a disc D_a is created from a random point from V_a and v_a. All points from V_a are processed until we encounter a point v_b for which $v_b \notin D_a$. Let V_b be the points processed from V_a thus far. We now know that points v_a and v_b are located on the boundary of the smallest enclosing disc of points V_b. Finding the smallest enclosing disc for a set of points is easy when two points on the boundary are known. First create a disc D_c from point v_a and v_b. Then traverse all points $v_c \in V_b$, if $v_c \notin D_c$ then update D_c with a new disc constructed from points v_a, v_b and v_c. This process of finding discs with 0, 1 and 2 points on the boundary known is repeated until we have found a disc that can contain all points.

The method is slightly adjusted such that the points that form the smallest enclosing disc is returned rather than the disc itself. For inputs containing at least 2 non-overlapping points the result is a set containing either 2 or 3 points.

Once the set of connector locations is know the partition parts can be supplemented with connectors. Two female holes are added to each geometry part located at each of the connector locations. The connector geometry itself is a custom geometry and can be changed according to the needs of the end user. A special closed edge loop of the connector geometry is marked. The vertices on these edges are all located at $z = 0$. When partitioning a geometry, the cross-section needs to be triangulated. Before this triangulation is applied the edge loops are inserted as holes to the cross-section polygon. After the cross section is triangulated the remainder of the cross-section geometry is connected to the holes to form the final geometry-part.

4 Results

Harpe partitions models to improve the parallel print time. An example of a partitioned and printed bunny model can be seen in fig. 24. Both models-parts were printed on a Ultimaker s5. The time spent printing for model a (fig. 24a) was 2 hours and the time spent printing model b (fig. 24b) was 2 hours and 5 minutes. In appendix A partitions for various models are depicted.



(a) Model part a

(b) Model part b

(c) Assembled model

Figure 24: Bunny model partitioned in $n = 2$ parts printed on two Ultimaker s5 printers.

A collection of around 400 models is used as input for the various methods. To ensure variety in the collection, the models are sourced from different repositories (as stated in section 7) and consist of CAD modelled mechanical parts, 3D scans and miniatures. These models are then converted[1] to the STL (Standard Tessellation Language) format. There are two variants when storing models in the STL format; a binary and an ASCII variant. Figure 25 describes the grammar of the ASCII variant of the STL file format.

$$
\begin{array}{l}
\textbf{solid } \textit{name} \\
\left\{\begin{array}{l}
\textbf{facet normal } n_i n_j n_k \\
\quad \textbf{outer loop} \\
\quad\quad \textbf{vertex } v1_x \ v1_y \ v1_z \\
\quad\quad \textbf{vertex } v2_x \ v2_y \ v2_z \\
\quad\quad \textbf{vertex } v3_x \ v3_y \ v3_z \\
\quad \textbf{endloop} \\
\textbf{endfacet}
\end{array}\right\}^{+} \\
\textbf{endsolid } \textit{name}
\end{array}
$$

Figure 25: Grammar of the STL file format.

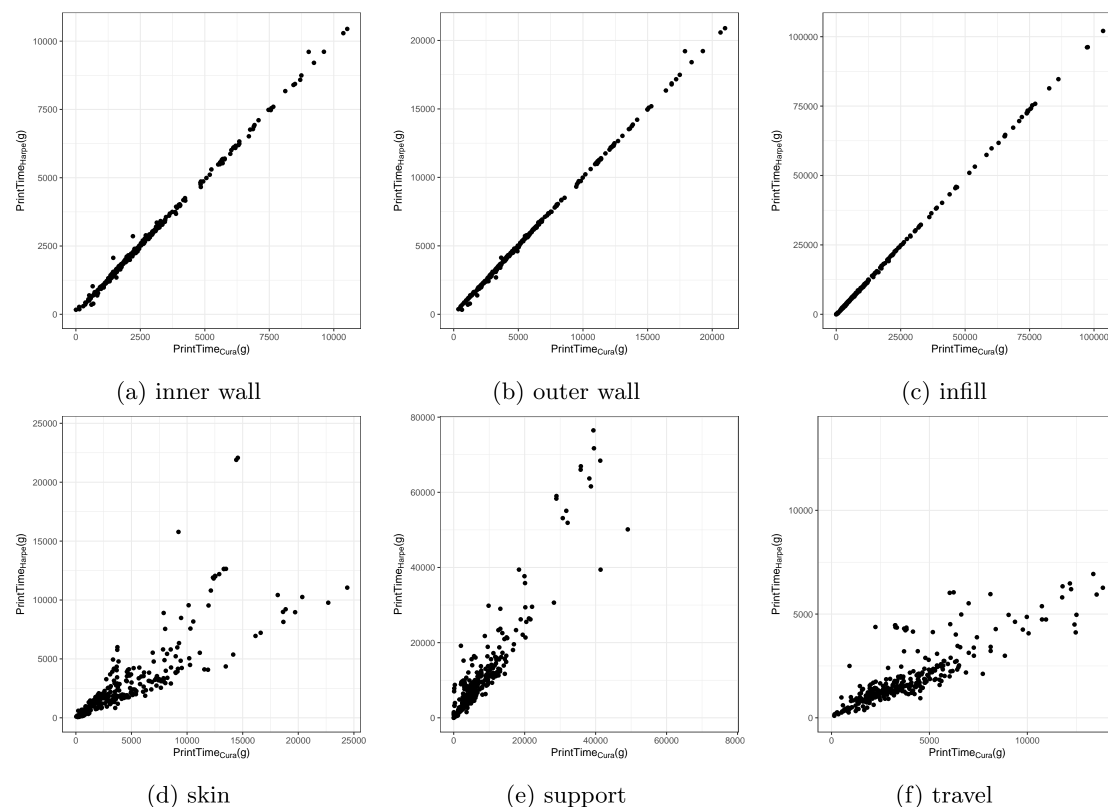
The STL files are then imported and all non-manifold and self-intersecting models are discarded. All models are adjusted such that the longest axis of the axis aligned bounding box is between 20 and 200 millimetres by scaling the model 10^x times where x is an integer.

Computing times reported are achieved using an intel i7 7700k CPU utilizing a single core.

4.1 Print Time Estimation

To evaluate the accuracy of Harpe's print time prediction the estimations are compared against the print times calculated by Cura. The assumption is made that Cura can accurately predict print times.

When estimating the print time each print-feature is estimated individually (fig. 26). For each feature a plot is generated where each measurement depicts the print time calculated by Cura on the x-axis and by Harpe on the y-axis. Some of these features can be predicted more accurately than others.



(a) inner wall (b) outer wall (c) infill

(d) skin (e) support (f) travel

Figure 26: Print time of each feature estimated by Harpe plotted against the print time calculated by Cura for 300 models.

The total print time of a model is the sum of the print times for each feature. Not each feature contributes as much to the total print time. Generally more time is needed to print the infill compared to the time spent traveling. The relative time spent per feature according to Cura is shown in table 1. The averages were calculated by adding the print time per feature of *all* models. The relative time spent printing per feature is calculated on these total print times.

Feature	Contribution to the total print time
skin	11.42%
inner wall	7.19%
outer wall	14.43%
infill	37.28%
support	19.85%
travel	9.83%

Table 1: Print time spent per feature.

The total print time estimated by HARPE is plotted against the total print time calculated by CURA in fig. 27. A histogram of the normalized print times is shown in fig. 28. The mean of the histogram is 0.974 with a standard deviation of 0.137.

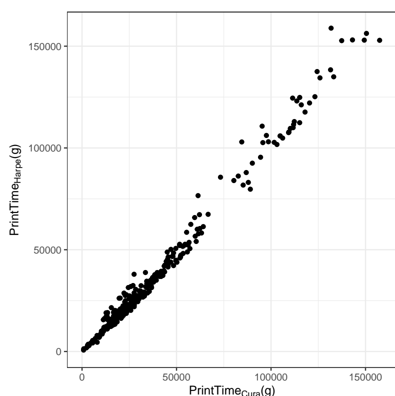


Figure 27: Total print time estimated by $\texttt{PrintTime}_{\text{HARPE}}$ plotted against the print time calculated by $\texttt{PrintTime}_{\text{CURA}}$ for 300 models.

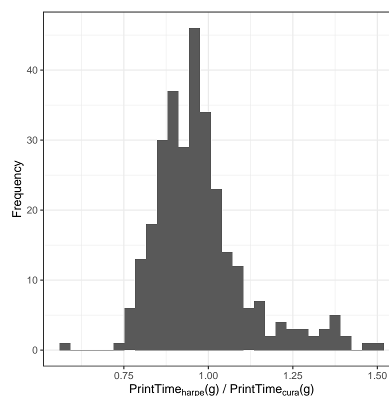


Figure 28: Histogram of percentage errors between the print time estimated using $\texttt{PrintTime}_{\text{HARPE}}$ and the print time calculated using $\texttt{PrintTime}_{\text{CURA}}$.

During the algorithm the above method for estimating the print time is *not* used but instead the simplified version where the print time is estimated using only the model's *volume* and *surface area*. The accuracy of this variant of the method is measured the same as before. The total print time estimated by HARPE is plotted against the total print time calculated by CURA in fig. 29. A histogram of the normalized print times is shown in fig. 30. The mean of the histogram is 0.976 with a standard deviation of 0.140.

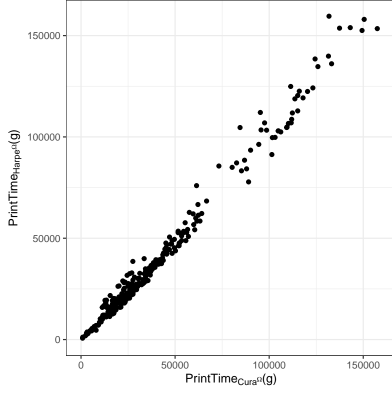


Figure 29: Total print time estimated by $\texttt{PrintTime}_{\texttt{HARPE}^\Omega}$ plotted against the print time calculated by $\texttt{PrintTime}_{\texttt{CURA}}$ for 300 models.

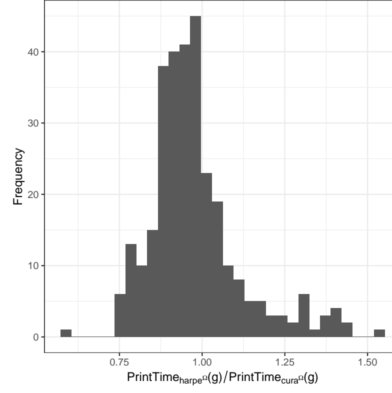


Figure 30: Histogram of percentage errors between the print time estimated by $\texttt{PrintTime}_{\texttt{HARPE}^\Omega}$ and the print time calculated by $\texttt{PrintTime}_{\texttt{CURA}}$.

As mentioned before the time complexity of the $\texttt{PrintTime}_{\texttt{HARPE}}$ function is linear in the number of triangles m. A plot depicting the computation time spent estimating the print time for models with triangle count m is shown in (fig. 31). The model with highest triangle count from this data set contains $m = 7,219,045$ triangles, the print time for this model was computed in 0.46 seconds.

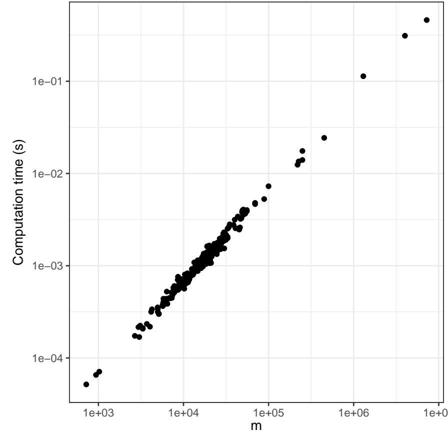


Figure 31: Computation time in seconds when calculating the print time for various models g with triangle count m.

For Cura the computation time for calculating the print time is in general bound by the size of the model, rather than the number of triangles. As the print time calculation is a by-product of creating the print instruction, the tool paths have to be generated. A bigger model results in a higher number of layers and tool paths. As the computation times for both Cura and Harpe are dependent on different factors it is complicated to compare these times. However when inspecting individual models it becomes apparent that Harpe is several orders of magnitude faster in estimating the print time compared to Cura as can be seen in table 2.

Model	m	Volume (mm^3)	Computation time of $\texttt{PrintTime}_{\texttt{HARPE}}$ (s)	Computation time of $\texttt{PrintTime}_{\texttt{CURA}}$ (s)
Stanford Bunny	69,664	770,056	$4.66 \cdot 10^{-3}$	33
Asian Dragon Low Poly Scan	249,881	154	$1.75 \cdot 10^{-2}$	7
Asian Dragon High Poly Scan	7,219,045	154	$4.61 \cdot 10^{-1}$	7
Neptune	4,007,872	12,578	$3.11 \cdot 10^{-1}$	33

Table 2: Computation time to calculate the print time for various models with triangle count m calculated by Harpe and Cura.

4.2 Partition Search

In order to compare the results of the Partition Search method with other algorithms the state-of-the art partitioning method Chopper is *transformed* into a parallel print time algorithm.

Chopper is originally designed to partition models such that each partition-part fits within a provided print volume. Additionally Chopper describes several objectives, a linear combination of these objectives determines the outcome of the partitioning. These objectives include objectives that favour symmetric cuts, and penalize fragile parts in the resulting decomposition. Our own implementation of Chopper is developed, and only the objectives that (in our opinion) would contribute to the *parallel print time partitioning* (the *part* and *utilization* objectives) were implemented.

The number of parts n is not an input parameter for Chopper, instead Chopper provides control over the maximum print volume. The transformation of Chopper consist of two parts: partitioning the model with the *aim* to partition the model in n parts, and selecting a print direction.

Chopper Partitioning In the results we want to report the print times of $n = 2, \ldots, 6$ parts. The relevant parameter Chopper provides is adjusting the print volume. There is a correlation between print volume and number of parts; partitioning models using a smaller print volume results in a higher number of parts. However this relation is not easily quantifiable. For each model partitioned using Chopper the longest axis of the axis aligned bounding box $aabb_{\max}$ was calculated. This value was then multiplied by a factor $f = 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6$. This $aabb_{\max} \cdot f$ is then used as the print volume. If the resulting BSP contains $n = 2, \ldots, 6$ parts the partitioning is incorporated in the results, otherwise the entry is discarded.

Selecting a print direction for Chopper The model-parts generated by Chopper are oriented such that each part fits within the print volume. However this orientation does not correspond to the orientation that minimizes the print time. For a fair comparison the model parts are oriented such that print time is taken into account. Similar as with the Partition Search each partition plane is a candidate print base. For each model-part the partition plane is chosen that minimizes the print time.

4.2.1 Comparison of search strategies

To accurately report the results, 400 models were partitioned by both Partition Search and Chopper. In order to compare different models with different print times the parallel print times are normalized. This is calculated by dividing the partition-part that takes the longest

to print by the print time of the original non-partitioned model. Results are calculated for 400 models partitioned in $n = 2, \ldots, 6$ parts. The normalized print time with a confidence interval of 95% is shown in fig. 32. Both PARTITION SEARCH and CHOPPER have parameters to control the density candidate cuts. These parameters, h to control the number of plane normals and τ the distance between planes, were set to 3 and 10.0 millimetre respectively. The print times were estimated using $\texttt{PrintTime}_{\texttt{CURA}}$.

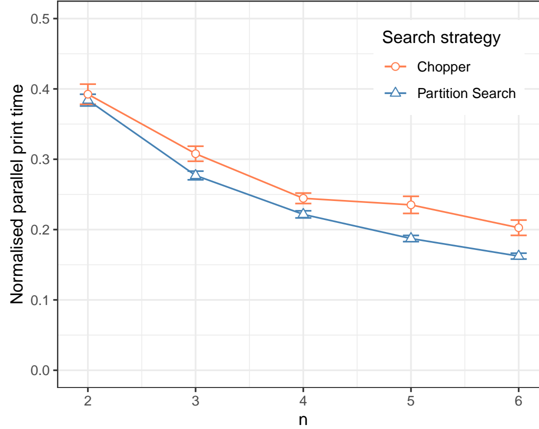


Figure 32: Confidence interval of 95% with 400 models depicting the normalized parallel print time of an $n = 2, \ldots, 6$ partitioned model using the search strategies PARTITION SEARCH and CHOPPER.

4.2.2 Performance of Partition Search

The performance of HARPE is compared against CHOPPER. The results of this comparison can be seen in fig. 33.

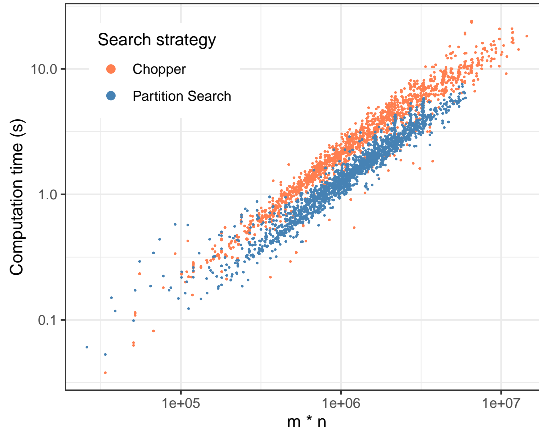


Figure 33: Computation time in seconds of PARTITION SEARCH (blue) compared to CHOPPER, for different models containing m triangles partitioned in n parts.

Note however that we did use our own implementation of CHOPPER, and as such the reported

performance might differ from the original implementation. After careful examination of the CHOPPER paper[6] we have identified several causes that might cause our implementation to differ from CHOPPER.

- CHOPPER stated that their implementation utilized multiple cores, where our implementation only uses one,
- Not all of the objectives described in CHOPPER were implemented, only those that we identified as being beneficial for the parallel print time partitioning were added, and
- CHOPPER benefits from the batched operations described in section 3.4. We were not sure if, and how many, of these batched operations were also implemented for CHOPPER.

4.3 Batched Calculations

As discussed in section 3.4 the batched calculations reduce the time complexity of the naive implementation from $O(mk)$ to $O(m + k + c)$, where m is the number of triangles of a geometry g, k the number of parallel planes in set P and c the number of plane-triangle intersections. The number of intersections c is bound by the number of planes $m \cdot k$; there are situations where it is possible for the naive algorithm to outperform the improved batched operations, however as we will see for all practical models this is not the case.

A set of planes spaced $\tau = 1$ millimetres apart, oriented in all three octagonal axis directions were used to perform the `Areas`, `CrossSectionalAreas`, `Volumes`, $\texttt{SupportVolumes}_{\texttt{geometry}}$ and $\texttt{SupportVolumes}_{\texttt{support}}$ operations on the data set. The results can be seen in fig. 34. The number of parallel planes k varies per model as not all models are the same size, and highly influences the computation time improved. For $k = 100$ a performance increase in excess of 100x can be achieved for some operations when batched compared to the naive implementation.

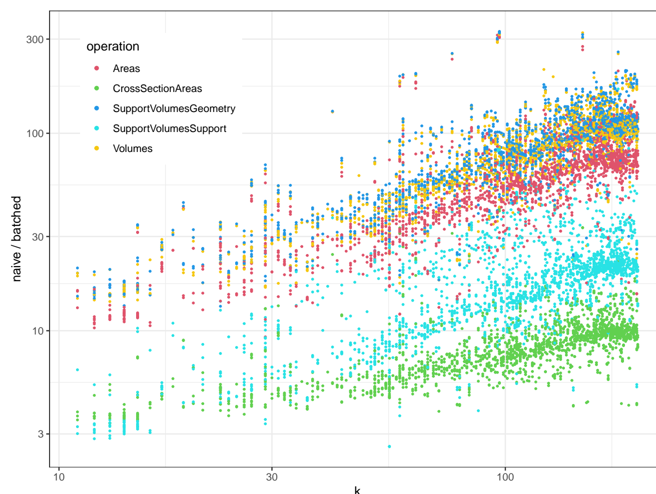


Figure 34: Computation time improvement for the *areas*, *cross section areas*, *support volumes geometry*, *support volumes support* and *volumes* batched operations as a function of the number of parallel planes k.

4.4 Local Search Procedure

The local search procedure is an iterative process that optimizes an existing BSP. After each iteration the planes in the BSP are moved such that the parallel print time is decreased. What became apparent is that the amount of optimization possible using this method depends on the parameter n; for a model partitioned in more parts, a greater decrease in parallel print time can be achieved. For this reason the results for different parameters n are reported independently.

Each model is partitioned into $n = 1, \ldots, 6$ parts using the PARTITION SEARCH method. Then the local search procedure is executed for $x = 1, \ldots, 30$ iterations. The parallel print time is calculated after each iteration. The parallel print time is normalized by dividing the parallel print time after iteration c by the original parallel print time. The results of a 95% confidence interval of these normalized parallel print times are shown in fig. 35.

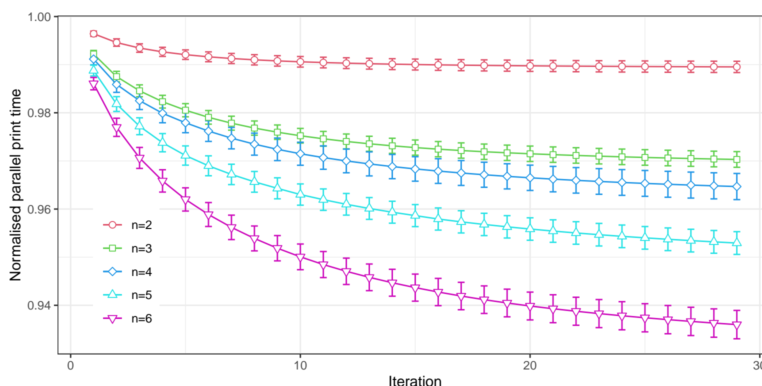


Figure 35: Confidence interval of 95% of the normalized improvement in parallel print time of 400 models partitioned in $n = 2, \ldots, 6$ parts for $x = 1, \ldots, 30$ iterations.

Computation time of the local search procedure is dependent on both the number of partition parts n and the number of triangles m in geometry g. Figure 36 shows a plot of the performance of a single iteration for various models g partitioned in $n = 1, \ldots, 6$ parts.

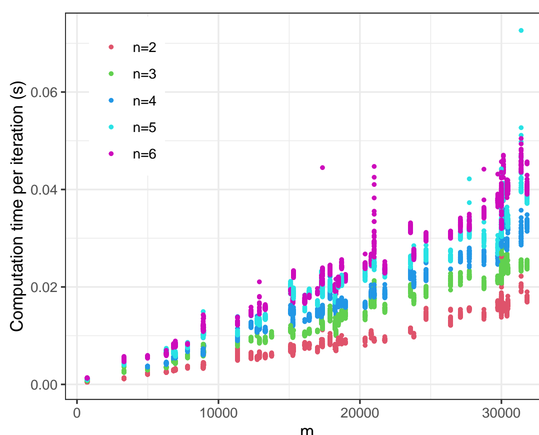


Figure 36: Computation time of local search procedure of a single iteration for a model g containing m triangles partitioned in $n = 1, \ldots, 6$ parts.

5 Conclusion

With HARPE we introduce a novel method for partitioning models to improve the parallel print time in Fused Filament Fabrication; PARTITION SEARCH. An input model g is partitioned in n (the number of available printers) parts. By partitioning a model some of the support structure becomes obsolete. PARTITION SEARCH favours decompositions that diminish the support volume, resulting in a parallel print time improvement that sometimes surpasses $\frac{1}{n}$ for lower values for n. On the other hand, by partitioning, the total print time of all models might increase due to the increase in combined surface area. For partitions with higher values n the impact of this increase in combined surface area becomes apparent. For higher values of n the parallel print time improvement slightly exceeds, but still approaches $\frac{1}{n}$ times the print time of the original model.

With the addition of the batched computations on the candidate cuts the algorithm becomes computationally more efficient. A larger number of cuts can be evaluated, resulting in a higher quality decomposition.

Compared to state-of-the-art partition method CHOPPER, HARPE can partition models in the required number of parts, produce model parts with improved parallel print time and is computationally faster.

With HARPE we introduce a novel method for estimating the print time of models. While less accurate compared to previous methods, it can predict the print times of models significantly faster. The proposed method takes into account the print settings used to print the model for the print time estimation. In addition to being a prerequisite for PARTITION SEARCH, regular slicers can also benefit from this method. Slicers such as CURA could for instance display material usage and print time *before* the model has been sliced. The material/time cost can be updated in real time after certain print settings have been changed by the user.

After a partitioning has been found it is improved by the local search procedure. Possible inefficiencies in the decompositions can be restored. The approach is versatile, any BSP partitioning could be refined using the local search procedure.

6 Discussion & Future work

Accuracy of the print time estimation The print time estimations of the inner wall, outer wall and infill features (fig. 26) can be estimated with high accuracy. As the volumes for these features are determined with high accuracy and printed continuously with a more or less constant speed, their print times estimation approximate the real print times well.

The print time estimations by Harpe of the skin, support and travel features (fig. 26) deviate more from their Cura counterparts. Print time spent on these features are inherently harder to predict. For each of the features this happens for a different reason.

- As mentioned in section 3.2 some liberties were taken when calculating support volume in order to maintain an efficient algorithm. The support volume is constructed from each surface that needs support downwards. Once the support is obstructed (by the model itself) there is no need for that support structure as the model itself provides sufficient support. Our approach (incorrectly) assumes all support is constructed from the print-base to the surface needing support.
- The travel times are inherently hard to predict. These travel times follow from a path optimization problem where the sequence of tool paths is found that minimizes the travel times. As these tool paths are never generated it is not possible to estimate the print times on these tool paths, but instead a heuristic approach based on the surface area of the model is taken to estimate this feature.
- The volume occupied by the skin feature would be easy to calculate were it not for the wall volume; if a region of space would be both filled by the skin and the wall volume then the wall feature takes precedence over the skin feature. To calculate the skin feature exactly we would need to know the volume of the intersection between the skin and wall regions. As this would be too expensive to calculate, some liberties were taken to estimate this volume.

In recent development researchers propose a novel tool path generation method where the tool paths are printed with variable width[20]. Harpe's approach for estimating the print time cannot easily be extended to account for tool-paths with variable line-width, as our approach assumes the flow rate to be constant for all features. This might however not be an issue as the researchers propose a different method to vary the line width; *back pressure compensation*. Here the flow rate remains constant and instead the travel speed is adjusted. As the flow rate remains the same the print time is not changed.

Implementation of Chopper The Chopper partitioning algorithm is implemented as a Beam Search objective. Beam Search improves greedy search methods by traversing the b most promising search branches. The beam width b should improve the quality of the partitioning, however as more branches are explored the running time of the algorithm grows. As Beam Search provides a flexible framework for defining custom objectives, the first version of Partition Search was implemented as such an objective.

For our research we started by exploring the Beam Search approach by defining a custom objective that evaluates partial BSP trees based on an estimate on the parallel print time. The reason for abandoning this approach was twofold. When defining a Beam Search objective for a parallelization objective only a lower bound on the resulting partitioning can be provided. As both the *average cross section heuristic* and *support volume* additions are upper bounds on the parallel print time, these two improvements could not be added to such an objective.

Additionally the parameter b did not appear to significantly improve the parallel print time; in some cases only a few percent was gained in the parallel print time when increasing b.

For our results we use our own implementation Chopper for comparison. As Chopper is also based on Beam Search the question arises if our implementation is correct; did the original authors encounter the same problem regarding the beam width b? We did not have access to the original Chopper implementation and we could only compare our results of Chopper against the results reported in their work. Unfortunately Chopper offers little data on these values in the results; only the objective value of two models were reported. The axis on which this objective was displayed showed only a small decrease in this objective (1.0995 to 1.0980 and 0.600 to 0.5596 for $b = 1$ and $b = 8$ respectively[6]). These values were in line with the minimal improvement we found when increasing b. We reached out to the authors of Chopper but received no further reply.

Future work Harpe provides an exploration in the field of partitioning models to improve the parallel print time. However, improvements are always possible. Future work could focus on

- when partitioning models using Partition Search it is assumed that all parts are printed with the same printer and print settings. Some people might have access to multiple printers, that might not be the same model. A possible improvement to Partition Search would be that a collection of different printers can be used as input,
- Harpe does not take into account a printer's volume; a possible extension to the algorithm would be to further partition pieces that exceed the print volume,
- additional partition criteria could be added to the partitioning function. Cuts that prevent narrow or fragile features in the resulting decomposition or hide cuts in creases could be favoured by such additional criteria, and
- an interesting subject not yet explored is how simplifying meshes affects the algorithm. If partitioning simplified models does not result in a significantly different outcome then the algorithm could be improved by partitioning only the simplified mesh. The resulting BSP can then be used to partition the high resolution model.

7 Acknowledgement

This research project was commissioned by 3D printer manufacturer Ultimaker B.V.[3] as a (paid) internship.

Models used to test, benchmark and generating the results presented throughout this thesis, were sourced from a repository called "A Benchmark for 3D Mesh Segmentation"[5], the AIM@SHAPE[21] repository and The Stanford 3D Scanning Repository[22].

The 3D Printing Handbook: Technologies, Design and Applications[14] is frequently consulted during the research. While not directly quoted in the work, the general knowledge gained from this book was of great benefit in understanding additive manufacturing.

[3]https://ultimaker.com/

A Partition Examples

Model	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
Dancing Children					
Dancing Children Oriented					
Fertility					
Fertility Oriented					
Kitten					
Kitten Oriented					

Table 3: Partitions for various models.

References

[1] Patrick Min. *meshconv*. Accessed: 2021-09-29. 1997. URL: http://www.patrickmin.com/meshconv.

[2] Cha Zhang and Tsuhan Chen. "Efficient feature extraction for 2D/3D objects in mesh representation". In: *Proceedings 2001 International Conference on Image Processing (Cat. No.01CH37205)*. Vol. 3. 2001, 935–938 vol.3. DOI: 10.1109/ICIP.2001.958278.

[3] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. Chap. 4.2, pp. 66–71. ISBN: 3540779736.

[4] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. Chap. 4.7, pp. 86–89. ISBN: 3540779736.

[5] Xiaobai Chen, Aleksey Golovinskiy, and Thomas Funkhouser. "A Benchmark for 3D Mesh Segmentation". In: *ACM Transactions on Graphics (Proc. SIGGRAPH)* 28.3 (Aug. 2009).

[6] Linjie Luo, Ilya Baran, Szymon Rusinkiewicz, and Wojciech Matusik. "Chopper: Partitioning Models into 3D-Printable Parts". In: *ACM Trans. Graph.* 31.6 (Nov. 2012). ISSN: 0730-0301. DOI: 10.1145/2366145.2366148. URL: https://doi.org/10.1145/2366145.2366148.

[7] Ruizhen Hu, Honghua Li, Hao Zhang, and Daniel Cohen-Or. "Approximate Pyramidal Shape Decomposition". In: *ACM Trans. Graph.* 33.6 (Nov. 2014). ISSN: 0730-0301. DOI: 10.1145/2661229.2661244. URL: https://doi.org/10.1145/2661229.2661244.

[8] J. Vanek, J. A. Garcia Galicia, B. Benes, R. Mundefinedch, N. Carr, O. Stava, and G. S. Miller. "PackMerger: A 3D Print Volume Optimizer". In: *Comput. Graph. Forum* 33.6 (Sept. 2014), pp. 322–332. ISSN: 0167-7055. DOI: 10.1111/cgf.12353. URL: https://doi.org/10.1111/cgf.12353.

[9] Xuelin Chen, Hao Zhang, Jinjie Lin, Ruizhen Hu, Lin Lu, Qixing Huang, Bedrich Benes, Daniel Cohen-Or, and Baoquan Chen. "Dapper: Decompose-and-Pack for 3D Printing". In: *ACM Trans. Graph.* 34.6 (Oct. 2015). ISSN: 0730-0301. DOI: 10.1145/2816795.2818087. URL: https://doi.org/10.1145/2816795.2818087.

[10] Peng Song, Zhongqi Fu, Ligang Liu, and Chi-Wing Fu. "Printing 3D objects with interlocking parts". In: *Computer Aided Geometric Design* 35-36 (2015). Geometric Modeling and Processing 2015, pp. 137–148. ISSN: 0167-8396. DOI: https://doi.org/10.1016/j.cagd.2015.03.020. URL: https://www.sciencedirect.com/science/article/pii/S0167839615000436.

[11] Kuo-Wei Chen, Chih-Yuan Yao, Yu-Chi Lai, and You-En Lin. "Parallel 3D Printing Based on Skeletal Remeshing". In: *ACM SIGGRAPH 2016 Posters*. SIGGRAPH '16. Anaheim, California: Association for Computing Machinery, 2016. ISBN: 9781450343718. DOI: 10.1145/2945078.2945126. URL: https://doi.org/10.1145/2945078.2945126.

[12] W. M. Wang, C. Zanni, and L. Kobbelt. "Improved Surface Quality in 3D Printing by Optimizing the Printing Direction". In: *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics*. EG '16. Lisbon, Portugal: Eurographics Association, 2016, pp. 59–70.

[13] Xiaotong Jiang, Xiaosheng Cheng, Qingjin Peng, Luming Liang, Ning Dai, Mingqiang Wei, and Cheng Cheng. "Models partition for 3D printing objects using skeleton". In: *Rapid Prototyping Journal* 23 (Jan. 2017). DOI: 10.1108/RPJ-07-2015-0091.

[14] Ben Redwood, Filemon Schffer, and Brian Garret. *The 3D Printing Handbook: Technologies, Design and Applications.* 1st. 2017. ISBN: 9082748509.

[15] Eric A. Yu, Jin Yeom, Cem C. Tutum, Etienne Vouga, and Risto Miikkulainen. "Evolutionary Decomposition for 3D Printing". In: *Proceedings of the Genetic and Evolutionary Computation Conference.* GECCO '17. Berlin, Germany: Association for Computing Machinery, 2017, pp. 1272–1279. ISBN: 9781450349208. DOI: 10.1145/3071178.3071310. URL: https://doi.org/10.1145/3071178.3071310.

[16] Xuelin Chen, Honghua Li, Chi-Wing Fu, Hao Zhang, Daniel Cohen-Or, and Baoquan Chen. "3D Fabrication with Universal Building Blocks and Pyramidal Shells". In: *ACM Trans. Graph.* 37.6 (Dec. 2018). ISSN: 0730-0301. DOI: 10.1145/3272127.3275033. URL: https://doi.org/10.1145/3272127.3275033.

[17] Xiangzhi Wei, Siqi Qiu, Lin Zhu, Ruiliang Feng, Yaobin Tian, Juntong Xi, and Youyi Zheng. "Toward Support-Free 3D Printing: A Skeletal Approach for Partitioning Models". In: *IEEE Transactions on Visualization and Computer Graphics* 24.10 (Oct. 2018), pp. 2799–2812. ISSN: 1077-2626. DOI: 10.1109/TVCG.2017.2767047. URL: https://doi.org/10.1109/TVCG.2017.2767047.

[18] E. Karasik, R. Fattal, and M. Werman. "Object Partitioning for Support-Free 3D-Printing". In: *Computer Graphics Forum* 38 (May 2019), pp. 305–316. DOI: 10.1111/cgf.13639.

[19] I. Filoscia, T. Alderighi, D. Giorgi, L. Malomo, M. Callieri, and P. Cignoni. "Optimizing Object Decomposition to Reduce Visual Artifacts in 3D Printing". In: *Computer Graphics Forum* 39.2 (2020), pp. 423–434. DOI: https://doi.org/10.1111/cgf.13941. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13941. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13941.

[20] Tim Kuipers, Eugeni L. Doubrovski, Jun Wu, and Charlie C.L. Wang. "A Framework for Adaptive Width Control of Dense Contour-Parallel Toolpaths in Fused Deposition Modeling". In: *Computer-Aided Design* 128 (2020), p. 102907. ISSN: 0010-4485. DOI: https://doi.org/10.1016/j.cad.2020.102907. URL: https://www.sciencedirect.com/science/article/pii/S0010448520301007.

[21] *Digital Shape Workbench - Shape Repository.* URL: http://visionair.ge.imati.cnr.it/ontologies/shapes/.

[22] *The Stanford 3D Scanning Repository.* URL: http://graphics.stanford.edu/data/3Dscanrep/.

[23] *Ultimaker Cura: Powerful, easy-to-use 3D printing software.* URL: https://ultimaker.com/software/ultimaker-cura.