COMPACT REPRESENTATION AND GENERATIVE DESIGN OF VORONOI BASED 3D INTERNAL STRUCTURES USING 2D SLICES IN ADDITIVE MANUFACTURING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ONAT AŞIK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
MECHANICAL ENGINEERING

JANUARY 2024

Approval of the thesis:

COMPACT REPRESENTATION AND GENERATIVE DESIGN OF VORONOI BASED 3D INTERNAL STRUCTURES USING 2D SLICES IN ADDITIVE MANUFACTURING

submitted by **ONAT AŞIK** in partial fulfillment of the requirements for the degree of **Master of Science in Mechanical Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. M. A. Sahir Arıkan
Head of Department, **Mechanical Engineering** ________________

Assoc. Prof. Dr. Ulaş Yaman
Supervisor, **Mechanical Engineering, METU** ________________

Examining Committee Members:

Prof. Dr. Melik Dölen
Mechanical Engineering, METU ________________

Assoc. Prof. Dr. Ulaş Yaman
Mechanical Engineering, METU ________________

Assoc. Prof. Dr. Ahmet Buğra Koku
Mechanical Engineering, METU ________________

Prof. Dr. Bahattin Koç
Industrial Engineering, Sabancı University ________________

Prof. Dr. Yusuf Sahillioğlu
Computer Engineering, METU ________________

Date: 25.01.2024

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Onat Aşık

Signature :

ABSTRACT

COMPACT REPRESENTATION AND GENERATIVE DESIGN OF VORONOI BASED 3D INTERNAL STRUCTURES USING 2D SLICES IN ADDITIVE MANUFACTURING

Aşık, Onat
M.S., Department of Mechanical Engineering
Supervisor: Assoc. Prof. Dr. Ulaş Yaman

January 2024, 101 pages

This study introduces a novel method for the efficient modeling, compact representation and additive manufacturing of 3D Voronoi based internal structures, utilizing 2D Voronoi slices. This innovative approach is underpinned by a newly proposed data format and a generative design methodology. The developed data format significantly reduces manufacturing data size for various geometries, offering a notable improvement over traditional data formats. The proposed method is engineered to process boundary representation data, alongside input parameters such as offset amount, Voronoi seed count, and their spatial distribution. It efficiently generates 2D Voronoi slices and NC files suitable for the Material Extrusion type Additive Manufacturing processes, leading to the successful fabrication of the output geometry. To construct 3D Voronoi structures from these 2D slices, the study introduces an innovative offsetting technique. This technique is based on the unique characteristics of the Weighted Voronoi Diagrams (also referred to as Power Diagrams) applied layer by layer. As part of this approach, new Voronoi seeds were strategically integrated to ensure a consistent Voronoi-to-Area ratio across the designated layers and coordinates. A significant outcome of this developed method is the elimination of overhang and the need

for support structures in the fabrication of 3D Voronoi structures, achieved through the controlled application of the offset parameter.

Keywords: Generative Design, Additive Manufacturing, Voronoi Diagrams, Compact Representation, Data Size Reduction

ÖZ

3 BOYUTLU VORONOİ TABANLI İÇ YAPILARIN 2 BOYUTLU DİLİMLER KULLANILARAK EKLEMELİ İMALATA YÖNELİK ÜRETKEN TASARIMI VE SADELEŞTİRİLMİŞ GÖSTERİMİ

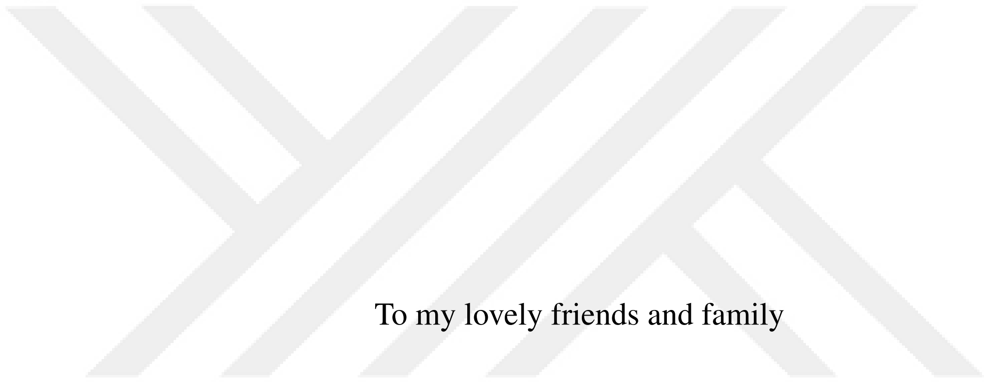
Aşık, Onat
Yüksek Lisans, Makina Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Ulaş Yaman

Ocak 2024 , 101 sayfa

Bu çalışmada, 2 boyutlu dilimler kullanılarak 3 boyutlu Voronoi tabanlı iç yapıların modellenmesi, sadeleştirilmiş şekilde gösterimi ve eklemeli imalat yöntemiyle üretimi için yenilikçi bir yöntem geliştirilmiştir. Sunulan yöntemle, üretken tasarım yöntemi kullanılarak yeni bir veri aktarım formatı ortaya konmuştur. Söz konusu veri aktarım formatı, çeşitli geometriler için geleneksel veri formatlarına göre üretim için gerekli veri boyutunu önemli ölçüde azaltmaktadır. Geliştirilen yöntemde, hedef parçanın sınır temsili verisi, ofset miktarı, Voronoi tohum sayısı ve bunların iki boyuttaki konumları girdi parametreleri olarak verilmektedir. Algoritmanın uygulanması sonucunda sonuç geometrisinin görsel gösterimi ve Malzeme Ekstrüzyonu ile eklemeli imalat yöntemlerine uygun NC dosyası çıktı alınmaktadır. NC dosyası çıktısı kullanılarak yapılan üretimlerle, önerilen yöntemle ortaya çıkan geometrilerin üretilebilirliği gösterilmiştir . 2 boyutlu dilimlerden 3 boyutlu Voronoi tabanlı iç yapıların oluşturulması için katman özelinde Ağırlıklı Voronoi Diyagramlarının özelliklerini kullanan bir ofsetleme yöntemi kullanılmaktadır. Oluşturulan katmanlar arasında tutarlı bir

Voronoi-Alan oranı sağlamak için yeni Voronoi tohumları belirlenen kurallara göre yeni katmanlara yerleştirilmektedir. Geliştirilen yöntemde ofset parametresinin kontrolü ile Voronoi tabanlı 3 boyutlu iç yapıların imalatında sarkma durumu engellenmiş, destek yapılarına duyulabilecek ihtiyaç ortadan kalkmıştır.

Anahtar Kelimeler: Üretken Tasarım, Eklemeli İmalat, Voronoi Diyagramları, Sadeleştirilmiş Gösterim, Veri Boyutu Küçültme



To my lovely friends and family

ACKNOWLEDGMENTS

I am deeply thankful to my family and friends. During this journey, there have been times when I couldn't spend as much time with them as I wished. Their understanding and endless support in the process have been invaluable. I sincerely appreciate their patience and constant encouragement.

I want to express my deepest gratitude to my academic advisor, Associate Professor Ulaş Yaman, for his constant support even when I lost hope and doubted my ability to succeed. Finishing this journey without his guidance and valuable contributions would be impossible.

I extend my sincere gratitude to Daniel González Abalde for creating and generously sharing his outstanding plug-in, SuperDelaunay. This tool has been invaluable in my thesis work, greatly contributing to my research.

I would also like to thank my colleagues at ASELSAN for their support.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xiv

LIST OF FIGURES xv

LIST OF ABBREVIATIONS xix

CHAPTERS

1 INTRODUCTION 1

1.1 Motivation and Problem Definition 1

1.2 Proposed Methods and Models 2

1.3 Contributions and Novelties 3

1.4 Limitations of the Proposed Method 4

1.5 The Outline of the Thesis 5

2 BACKGROUND INFORMATION & LITERATURE REVIEW 7

2.1 Background Information 7

2.1.1 2D Voronoi Diagrams 7

2.1.2 Power Diagrams 10

2.1.3 Super Delaunay Plug-In 11

2.2 Literature Review 13

2.2.1 Design for Additive Manufacturing 13

2.2.2 Generative Design 14

2.2.3 Use of Internal Structures in Additive Manufacturing 15

2.2.4 Efficient Modelling and Representation Methods of Internal Structures in Additive Manufacturing 16

2.2.5 Applications of Voronoi Diagrams 19

3 MODELING, COMPACT REPRESENTATION AND ADDITIVE MANUFACTURING OF 3D VORONOI BASED INTERNAL STRUCTURES . . 23

3.1 Input Parameters and Pre-Processing for Voronoi3DGenerator Plug-In 25

3.1.1 Conversion from Mesh to BRep 25

3.1.2 Centering of the BRep at the Cartesian Origin 26

3.1.3 Slicing of the Translated BRep 28

3.1.4 Generation of Voronoi Seed Points 30

3.2 Modeling of 3D Voronoi Structures using Voronoi3D Generator Plug-In 30

3.2.1 3D Voronoi Generator Algorithm 32

3.2.2 Proximity Check Between Polyline Points and a Curve Algorithm 39

3.2.3 Precision Adjustment of 3D Point Coordinates Algorithm . . . 39

3.2.4 Distinct Point Enumeration in a Geometric Polyline Algorithm 39

3.2.5 Symmetric Index Sequence Generation from a Voronoi Structure Algorithm 40

3.2.6 Point Insertion Algorithm 40

3.2.7 Optimization of Weights in Voronoi Structure Generation Algorithm . . . 42
3.2.8 Post-Offset Algorithm for Inserted Cells . . . 43
3.3 Visualization of Output Geometry . . . 45
3.4 Custom Data Format Generation and Application . . . 46
3.5 Manufacturing of Output Geometry . . . 48
3.5.1 Elimination of Duplicate Polylines . . . 49
3.5.2 Continous Path Generation for Fabrication . . . 49
3.5.3 NC File Generation for FFF Systems . . . 50
3.5.4 NC File Manipulation . . . 52
3.5.5 Time Complexity Analysis . . . 55
3.5.6 2D Linear Interpolation . . . 56
4 IMPLEMENTATION & TEST CASES . . . 59
4.1 Results and Visualizations for Voronoi3DGenerator . . . 59
4.2 Results for 2D Linear Interpolation . . . 71
5 DISCUSSIONS & CONCLUSION . . . 73
REFERENCES . . . 79
APPENDICES
A PSEUDOCODES OF ALGORITHMS . . . 85

LIST OF TABLES

TABLES

Table 2.1 Advantages and Disadvantages of Data Formats in AM 17

Table 3.1 List of Variables 31

Table 3.2 Fabrication Parameters for 3D Printing 51

Table 4.1 Comparative Results of File Sizes Based on Input Parameters for **Voronoi3DGenerator** Plug-In 60

Table 4.2 Comparative Results of Execution Times Based on Input Parameters for Voronoi3DGenerator Plug-In 71

LIST OF FIGURES

FIGURES

Figure 2.1 Voronoi Diagram with Euclidian Distance Function 8

Figure 2.2 Voronoi diagrams generated with different distance function: **(a)** L2-metric distance; **(b)** Manhattan distance; **(c)** Chebyshev distance; **(d)** Chebyshev distance with axial scale; **(e)** L2-metric with axial scale; and **(f)** equidistant line distance. The partitioned regions are presented in different colors. [1] 9

Figure 2.3 A Voronoi tessellation for 400 random sites (in gray), bounded by a cube. Voronoi cell edges are shown in red, Delaunay edges in blue [2]. 9

Figure 2.4 Representation of 2D Laguerre Tessellation. Cyan circles are the generators with their weights, red lines are the resulting tessellation [3] 10

Figure 2.5 A Voronoi diagram generated with *Super Delaunay* for 10 randomly inserted seeds 12

Figure 2.6 A Voronoi diagram generated with *Super Delaunay* for 10 randomly inserted seeds with weight asserted to the cell C_4 12

Figure 2.7 A Voronoi diagram generated with *Super Delaunay* for 10 randomly inserted seeds with weight asserted to cells C_4 and C_7 13

Figure 2.8 Voronoi Diagrams in Nature [4] 21

Figure 3.1 Overall Workflow of the Process 24

Figure 3.2 Grasshopper Workflow 25

Figure 3.3 Mesh to BRep Conversion in *Grasshopper Canvas* 26

Figure 3.4 Schematic of Centering of the BRep at Cartesian Origin in Grasshopper 27

Figure 3.5 Centering Operation of Truncated Pyramid Mesh at the Cartesian Origin 29

Figure 3.6 Slicing Operation of the Translated BRep in Grasshopper 30

Figure 3.7 Initial Layer Seed Points Generation in Grasshopper 30

Figure 3.8 3D Voronoi Generator Algorithm Workflow 33

Figure 3.9 Minimum Spanning Tree Generation and Weight Assertion for a Truncated Pyramid 35

Figure 3.10 Voronoi3DGenerator Plug-In 36

Figure 3.11 **Voronoi3DGenerator** Result for Layer 60 37

Figure 3.12 **Voronoi3DGenerator** Result for Layer 190 38

Figure 3.13 Results of *Density List (DL) for the Example Application* 38

Figure 3.14 Visualization of Target Layer Before Point Insertion 41

Figure 3.15 Visualization of Target Layer After Point Insertion 42

Figure 3.16 Visualization of Post-Offset Operation 45

Figure 3.17 Preview Gate Block 45

Figure 3.18 Preview Interface 46

Figure 3.19 Export File C# Block 47

Figure 3.20 Output Data Format Structure 47

Figure 3.21 Import File C# Block 48

Figure 3.22 Generated Path Using Continuous Path Generation Algorithm . . 50

Figure 3.23 Example NC File Processor Block 52

Figure 3.24 NC File Generator Block 52

Figure 3.25 Simulation of the Original NC File in Cura Software 53

Figure 3.26 Segment of the Original NC File 53

Figure 3.27 Simulation of the Manipulated NC File in Cura Software 54

Figure 3.28 Segment of the Manipulated NC File 55

Figure 3.29 NC File Manipulator Block 55

Figure 3.30 Number of Voronoi Seeds vs Execution Time Graph 56

Figure 3.31 2D Linear Interpolation Pre-Processing Blocks 57

Figure 3.32 2D Linear Interpolation and Verification Blocks 57

Figure 4.1 Top View of Generated Cone 61

Figure 4.2 Perspective View of Generated Cone 61

Figure 4.3 Top View of Fabricated Cone 62

Figure 4.4 Bottom View of Fabricated Cone 62

Figure 4.5 Perspective View of Fabricated Cone 63

Figure 4.6 Top View of Generated Truncated Pyramid 64

Figure 4.7 Perspective View of Generated Truncated Pyramid 64

Figure 4.8 Top View of Fabricated Truncated Pyramid 65

Figure 4.9 Perspective View of Fabricated Truncated Pyramid 65

Figure 4.10 Top View of Generated Large Truncated Pyramid 66

Figure 4.11 Perspective View of Generated Large Truncated Pyramid 66

Figure 4.12 Top View of Fabricated Large Truncated Pyramid 67

Figure 4.13 Perspective View of Fabricated Large Truncated Pyramid 67

Figure 4.14 Top View of Generated Stanford Bunny 68

Figure 4.15 Perspective View of Generated Stanford Bunny 68

Figure 4.16 Bottom View of Fabricated Stanford Bunny 69

Figure 4.17 Perspective View of Fabricated Stanford Bunny 69

Figure 4.18 Top View of Generated Hypothetical Rectangular Prism 70

Figure 4.19 Perspective View of Generated Hypothetical Rectangular Prism . 70

Figure 4.20 Top View of Interpolated Cube 71

Figure 4.21 Perspective View of Interpolated Cube 72

Figure 5.1 Failure Case where Discontinuity Occurs 76

LIST OF ABBREVIATIONS

ABBREVIATIONS	
2D	Two Dimensional
3D	Three Dimensional
AM	Additive Manufacturing
AMF	Additive Manufacturing File
BRep	Boundary Representation
DfAM	Design for Additive Manufacturing
FDM	Fused Deposition Modeling
FFF	Fused Filament Fabrication
MFG	Manufacturing
NC	Numerical Control
STL	Standard Triangle Language

CHAPTER 1

INTRODUCTION

1.1 Motivation and Problem Definition

Through the utilization of Additive Manufacturing (AM) methods, it is possible to fabricate complex parts with internal structures. Among these internal structures, Voronoi diagrams have been recognized. The Voronoi diagram is a data structure extensively investigated within the scope of computational geometry applications [5]. Due to their mathematical properties and the capability to partition space into regions based on proximity to a defined set of points, Voronoi diagrams have been employed in various application fields such as mechanical engineering, civil engineering, architecture, biomedical sciences, geosciences, electronics, etc.

Fabrication of parts filled with 3D Voronoi based internal structures is mainly performed with AM methods due to the complex shapes of Voronoi structures. Challenges arise when Voronoi structures diminish in size and the target parts to be filled expand. Modeling and representing the 3D Voronoi based internal structures become more challenging. In this context, the DARPA TRADES competition [6] exemplifies the profound challenges faced by current computational methods in accurately representing shape and material properties of internal structures at extensive scales and in highly complex scenarios. The competition outlines a specific challenge: calculating integral properties, such as mass, and generating slicing profiles suitable for 3D printing a one cubic meter volume cube, filled with complex internal micro-truss structures with irregular connections at a micro-scale of 0.1mm. This task underscores the complexities involved in the modeling, representation, and additive manufacturing of large-scale and highly complex internal structures, mirroring the issues

encountered in working with 3D Voronoi based internal structures.

As in the case similar to DARPA competition, the modeling and generation of representation outputs for Voronoi structures can become time-intensive, and data sizes might escalate to terabytes levels, especially with formats based on Boundary Representation (BRep) such as STL and AMF. Since a resolution exists while exporting BRep based formats, the exactness of the geometry may not be preserved. Moreover, the fabrication process of parts filled with 3D Voronoi based internal structures may be challenging since overhanging surfaces may create the risk of failure during fabrication. These regions may require support structures.

1.2 Proposed Methods and Models

To overcome the problems stated in Section 1.1, a novel methodology is proposed for efficient modeling, compact representation and AM of 3D Voronoi based internal structures. The method is developed in *Rhinoceros 7* Software *Grasshopper* module. A custom Grasshopper Plug-In, named **Voronoi3DGenerator** is developed in Visual Studio 2019 by using C# for efficient modeling combined with proposed custom .txt based data format named **.ont** for compact representation. In addition, the necessary NC files are generated for the output geometry and then manufactured using Fused Filament Fabrication (FFF) type of AM machineries. As an advantage of the proposed methodology, the support need and overhang risk for Voronoi structures over the produced layers are also eliminated.

The STL data of the target part is imported into *Rhinoceros* software and the user sets the input parameters *Number of Voronoi Seeds*, *offset* for enlargement and contraction of Voronoi structures over the layers and *layer height* for AM. A pre-processing is applied to prepare input for **Voronoi3DGenerator** Plug-In using built-in and external Grasshopper blocks.

The modeling process is performed in **Voronoi3DGenerator** Plug-In to ensure constant Voronoi density along the layers. The built-in Grasshopper functions as well as the functions of external Plug-Ins such as *SuperDelaunay* and *Clipper* are used in the development process and the target part filled with 3D Voronoi based structures is

taken as an output.

The output geometry is represented via an NC file for the fabrication by using the external Plug-In, *Droid*. Since there don't exist any built-in machine settings in *Droid*, the output NC file is manipulated by using an example NC file, generated by the standard slicer software with built-in machine settings, given as an input by using a custom script. By this way, commands which are specific to the target FFF machine are included in the generated NC file. The fabrication of the output geometry is performed by using FFF machines.

For the representation, text based file format **.ont** which includes *Voronoi Seed Point coordinates*, *offset* and *layer height* is proposed. The data size to represent the output file is significantly reduced compared to BRep based representation formats such as STL and AMF. The output representation file, **.ont**, is imported to the program and a geometry exactly the same as the output geometry is obtained, ensuring the proposed data format is suitable for the data transfer process.

1.3 Contributions and Novelties

The contributions of the proposed methodology are as follows:

- A new method is proposed for the generative design of 3D Voronoi based internal structures from 2D Voronoi slices.
- Given an STL file, the part is filled with 3D internal Voronoi structures with the developed Grasshopper Plug-In called **Voronoi3DGenerator**.
- A novel data format called **.ont** is proposed which provides an efficient representation and significant data size reduction over conventional BRep data formats for 3D Voronoi based internal structures. Moreover, the exactness of the geometry is fully preserved in the data transfer process.
- Overhang and support requirement in the fabrication of 3D Voronoi based internal structures is eliminated by using controlled offset parameter.

- NC file is generated for the output geometry and fabrication is performed using an FFF machine. The manufacturability without the need for support structures is proven.

1.4 Limitations of the Proposed Method

This section delineates the constraints and limitations of the proposed methodology.

- While the developed method successfully prevents the requirement for support structures within the internal Voronoi configurations, the boundary representation (BRep) of the geometry may present overhanging regions necessitating support. These requisite structures for fabrication, particularly in the context of Fused Filament Fabrication (FFF), are currently beyond the method's capacity to generate.
- In the modeling process of internal 3D Voronoi based internal structures using **Voronoi3DGenerator**, discontinuities occur during the post-offset process, leading to the necessity of preserving the original vertex count in the inserted cells. Consequently, these cells cannot initiate from a minimal size and expand continuously to eliminate concavity over the layers, a limitation not encountered in direct 3D Voronoi diagram generation. Additionally, there is an observed phenomenon where cells sometimes vanish despite not being small, which can lead to bridging challenges in subsequent layers, particularly when lines above these regions are elongated.
- For FFF, the continuity of the path followed by the nozzle is crucial for minimizing residual materials. While a continuous path algorithm is implemented, it does not achieve complete continuity. Although most fast travel movements occur along the printed Voronoi edges, there are instances of rapid travel between distant locations, which is sub-optimal for the manufacturing process.

These limitations underscore areas for potential enhancement and refinement in future iterations of the method.

1.5 The Outline of the Thesis

Chapter 2 provides foundational background information essential for a comprehensive understanding of the proposed methodology. Additionally, this chapter reviews related works in the literature. Chapter 3 includes a detailed explanation of the proposed method, including explanations and illustrations of utilized algorithms and Grasshopper blocks. In Chapter 4, visualization and fabrication results from various test cases are demonstrated. This chapter also includes a comparative analysis of the data sizes between the proposed **.ont** data format and traditional BRep format, STL. Chapter 5 articulates the conclusions derived from the study, highlighting the principal findings, their implications, and paths for future research. Finally, Appendix A includes the pseudocodes of the algorithms explained in Chapter 3.

CHAPTER 2

BACKGROUND INFORMATION & LITERATURE REVIEW

This chapter gives the necessary background information and critical concepts about the thesis. Furthermore, related work to the presented study is discussed.

2.1 Background Information

This section includes comprehensive background information regarding the terminology employed in the proposed methodology. As the proposed methodology is mainly based on Voronoi diagrams, the definition and construction of Voronoi diagrams are explained in detail. Additionally, the external *Grasshopper* Plug-In, *Super Delaunay*, which plays a crucial role in the development of the **Voronoi3DGenerator** Plug-In, is described, supplemented by application examples to illustrate its functionality and integration.

2.1.1 2D Voronoi Diagrams

Voronoi diagrams, widely used data structures in computational geometry, are the minimization diagram of a finite set of continuous functions. These functions, generally used as the distance functions to the specified object, can differ with the type of application. The Voronoi diagram subdivides the embedding space into regions where each region includes points closer to a given object than the others. Many variants of the Voronoi diagrams have been defined considering the class of objects, the distance functions and the embedding space [7].

Georgry Voronoi has defined the Voronoi diagrams as given a set of points $p_1, p_2, ...p_n$

in 3D space, called sites, the corresponding Voronoi tessellation is a set of cells C_i where each cell consists of all the points p at least as close to p_i as to any other site [8]. The corresponding Delaunay triangulation [9] is a graph created by placing a straight edge between any two sites that share a cell boundary in the Voronoi tessellation. Thus, every site is connected to its nearest neighbors. This can be considered as a general definition of Voronoi diagrams where Euclidian distance is used as a distance function.

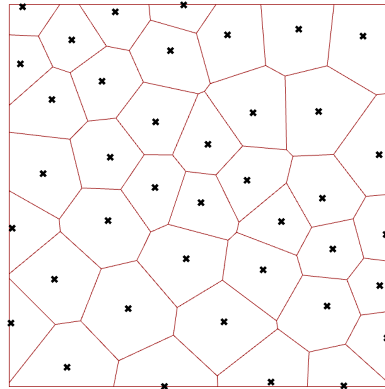


Figure 2.1: Voronoi Diagram with Euclidian Distance Function

While Voronoi diagrams are used in various applications, it's notable that there exist applications where some modifications can be beneficial and practical compared to original Voronoi diagrams. Objects other than points, different distance functions, and higher order diagrams are utilized specifically considering the required application [10].

Some variations of Voronoi diagrams are as follows:

- Power Diagram (Laguerre Diagram)
- Voronoi Diagrams with Manhattan Distance Functions
- Farthest-Point Voronoi Diagrams
- Approximate Voronoi Diagrams

Figure 2.2 illustrates the example Voronoi diagrams constructed with different distance functions.

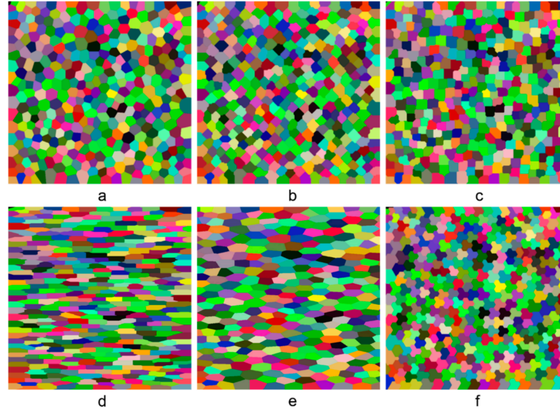


Figure 2.2: Voronoi diagrams generated with different distance function: **(a)** L2-metric distance; **(b)** Manhattan distance; **(c)** Chebyshev distance; **(d)** Chebyshev distance with axial scale; **(e)** L2-metric with axial scale; and **(f)** equidistant line distance. The partitioned regions are presented in different colors. [1]

Figure 2.3 presents a representative example of Voronoi tessellation in 3D. In this tessellation, each Voronoi cell is bounded by a convex polyhedron. A key feature, known as a Delaunay edge, a line segment linking two sites that share a polygonal face, is bisected perpendicularly by the plane of the face, though it is notable that the bisection point may not always reside within the face itself. Interestingly, when sites are selected randomly from a uniform distribution, the average number of nearest neighbors (which is equivalently the average number of cells sharing a face with any given cell) reaches an expectation value of approximately 15.54 [2].

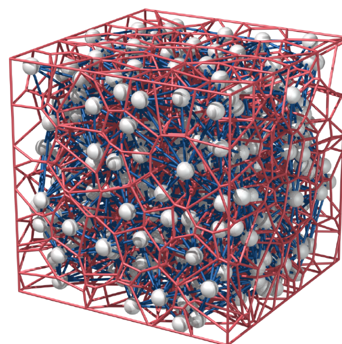


Figure 2.3: A Voronoi tessellation for 400 random sites (in gray), bounded by a cube. Voronoi cell edges are shown in red, Delaunay edges in blue [2].

2.1.2 Power Diagrams

Power diagrams, also called *Laguerre diagram* of *Dirichlet cell complex*, have been studied for a long time in mathematics and other areas of science. The power function was mentioned as a generalized distance function by Laguerre and Voronoi. Power diagrams play an important role in packing and covering spheres and illuminating balls and are applicable to particular problems in number theory.

The power $pow(x, s)$ of a point x with respect to a sphere s in Eucladian d-space E^d is given by $d^2(x, z) - r^2$ where d denotes Euclidean distance function, and z and r are the center and the radius of s. The power diagram of a finite set S of spheres in E^d is a cell complex that associates each $s \in S$ with convex domain $\{x \in E^d | (pow(x, s) < pow(x, t)$, for all t $\in S - \{s\}\}$ [10].

An example illustration of a power diagram can be seen in Figure 2.4.

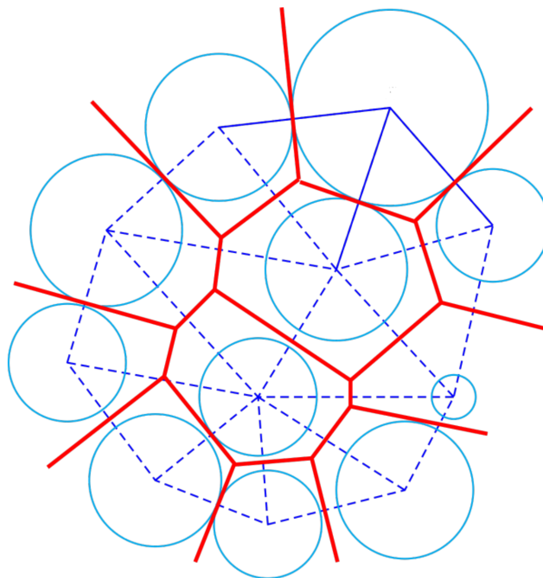


Figure 2.4: Representation of 2D Laguerre Tessellation. Cyan circles are the generators with their weights, red lines are the resulting tessellation [3]

2.1.3 Super Delaunay Plug-In

The external Plug-In, *Super Delaunay* developed by Daniel González Abalde [11], is utilized for the construction of Weighted Voronoi Diagrams, Minimum Spanning Tree and Neighbouring Trees in the **Voronoi3DGenerator** Plug-In presented in Section 3.2.1.

In the Plug-In source code, the distance function for Voronoi diagrams is defined as power metric and power diagrams are obtained. Firstly, Delaunay triangulation is performed for the given set of points. For the Delaunay triangulation, a modified version of the Bowyer-Watson algorithm is used. Delaunay triangulation is employed in the construction of weighted Voronoi diagrams.

As stated in Section 2.1.1, the choice of distance functions utilized in constructing Voronoi diagrams varies depending on the application. In *Super Delaunay* Plug-In, a distance function is used, the same as the power diagrams based on circles. Consequently, assigning weight values to each seed point becomes significant, thereby outlining their relative significance. Points with higher weight values are consequently represented as larger in size, reflecting their increased influence within the diagram's structure.

The distance function for point p with respect to the target Orthoball is $d^2(p, z) - r^2 - w(p)$; where d is the Euclidian distance function, z is the center of the Orthoball, r is the radius of the Orthoball and $w(p)$ is the weight of point p.

Figure 2.5 showcases an example of a Voronoi diagram generated with ten seed points (sites) confined within a square boundary curve. In this illustration, all sites are assigned a zero weight value. The indices corresponding to each Voronoi site are also prominently displayed for reference.

For the Voronoi cell C_4, a weight value of 50 is asserted and it's observed that the cell expands where the neighboring cells go under contraction due to the nature of the Power diagram. The Voronoi diagram is illustrated in Figure 2.6.

Now, the same weight value, 50, is asserted to the neighboring cell C_7. It can be seen that C_7 expands while the boundary edge between C_4 and C_7 turns back to the

position at the original diagram shown in Figure 2.5. This phenomenon stems from the equalization of weight values between the two cells, signifying a lack of relative significance or dominance of one cell over the other. Consequently, their mutual boundary reverts to its initial position observed in the original Voronoi diagram. This characteristic is crucial in the developed algorithms explained in Chapter 3 to preserve the convexity of 3D Voronoi structures. The aforementioned case is illustrated in Figure 2.7.

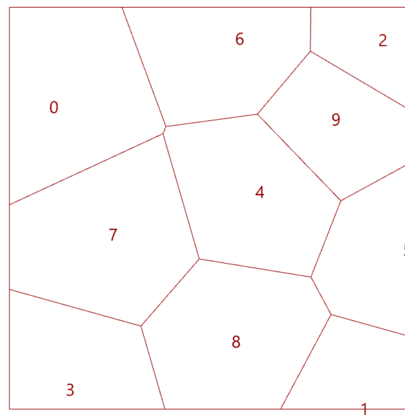


Figure 2.5: A Voronoi diagram generated with *Super Delaunay* for 10 randomly inserted seeds

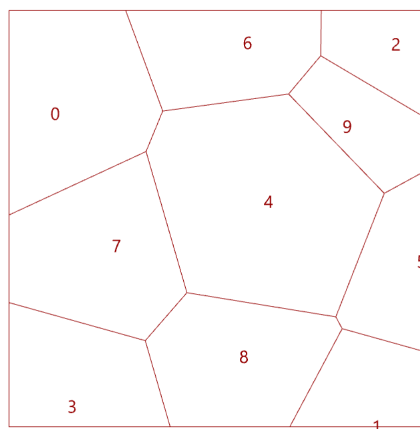


Figure 2.6: A Voronoi diagram generated with *Super Delaunay* for 10 randomly inserted seeds with weight asserted to the cell C_4

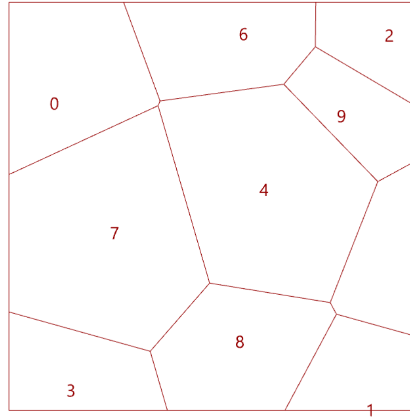


Figure 2.7: A Voronoi diagram generated with *Super Delaunay* for 10 randomly inserted seeds with weight asserted to cells C_4 and C_7

2.2 Literature Review

A literature review regarding the topics related to the research is presented in the following subsections.

2.2.1 Design for Additive Manufacturing

Design for Additive Manufacturing (DfAM) is a design methodology used while designing parts to be manufactured by AM methods to exploit the benefits of AM in product innovation and manufacturing [12]. It's also defined as "the use of design thinking to help identify, validate and communicate high-value propositions enabled by AM [13]. By applying DfAM methods, it's possible to use the design freedom of AM and create complex geometries while ensuring the manufacturability of parts to be fabricated using AM processes [14]. Moreover, DfAM provides unique advantages, including complete control over lattice structures, programmatic capability, and the capability to output both STL files and associated data for numerical analysis [15].

It is crucial for designers to comprehend the distinct design constraints included in DfAM, as these constraints vary across different AM methods [16]. The rules, guide-

lines, and tools of DfAM need to be thoroughly understood and accurately applied by designers. Generally, three principal tools are employed in DfAM methodologies: Topology Optimization, Lattice Generation, and Generative Design which can be combined to improve the effectivity of DfAM [17].

Topology optimization is a computational method to find the optimum material distribution in the defined design space under the specified boundary conditions. The extra material in the design space is removed; therefore, the weight of the target part is reduced while ensuring the required performance criteria are still satisfied [18]. This method is highly preferable in the application areas where weight reduction becomes a crucial parameter such as aerospace, mechanical and civil engineering. AM methods are generally highly suitable for the fabrication of the output geometry since the output geometry can be complex and challenging to manufacture by utilizing conventional manufacturing methods.

Lattice structures are another significant tool in DfAM for improving specific properties of the target part, such as mechanical and thermal properties. These structures exhibit a repetitive pattern that stems from interconnected struts, beams, and implicit functions [19]. These structures reduce material usage while improving mechanical properties and are particularly utilized by AM methods due to their geometric complexity. [20]. They find extensive application in the fields of aerospace, automotive, and medical implants due to these advantageous properties [19].

Further exploration of Generative Design as a tool in DfAM is presented in the Subsection 2.2.2.

2.2.2 Generative Design

In the research conducted by Han et al. [21], cross joints, crucial components in structural engineering applications, have been designed using machine learning-based generative design algorithms. Most generatively designed joints show novel configurations that exceed the conventional bounds of design imagination, typically based on experience. From several hundred automatically generated design alternatives, three were selected considering the design objectives and cost efficiency for cross joints.

These selections underwent further numerical analyses to identify the most suitable configuration for the desired static behaviors.

The output geometries have been manufactured using AM methods to avoid traditional manufacturing limitations. The generatively designed joints demonstrated superior mechanical performance while being lighter and more aesthetically appealing than conventional cross-joint designs. Therefore, it's indicated that generative design can be a powerful tool for finding the optimal and creative design alternatives.

In their research, Isakhani et al. [22] employed a generative design approach for a gliding wing, which assists the airplane pilot in maintaining the attitude of the glider and controlling lift and drag. While the shapes of natural wings have evolved over millions of years, artificial wing designs have been traditionally constrained to human-conceived, sub-optimal conceptualizations. Therefore, a generative design approach combined with artificial intelligence generates mechanically improved gliding wings. This approach involves generating multiple performance-driven solutions (wings) aligned with high-level objectives, utilizing an infinite-scale cloud computing solution executing a machine learning-based generative design algorithm. The most promising Computer-Aided Solutions for Design concepts are subjected to rigorous numerical analysis, fabrication, and mechanical testing. These outcomes are compared against existing literature for qualitative and quantitative analysis and validation. It's concluded that generatively designed tandem wings show 78% increased performance regarding withstanding fracture failure compared to the conventional models. On the other hand, the weight is increased by 11% and stiffness decreased by 14%.

2.2.3 Use of Internal Structures in Additive Manufacturing

AM is an enhanced manufacturing method to produce parts layer by layer based on the designed model and has numerous advantages over the conventional manufacturing methods and its application is spread in various fields. [23]

Internal structures are one of the topics mainly focused on AM. Since the part is produced layer by layer and not machined on its surface, it's possible to produce internal

structures in parts that enhance performance, mechanical properties and functionality [24]. The strength, stiffness and fatigue resistance of the components can be improved by the usage of the specific AM methods [25]. The functionality enhancements include improving fluid properties by producing fluid flow channels, which are challenging to produce by conventional methods. Also, the electrical pathways can be directly integrated into the parts during manufacturing in specific AM methods [26].

Material usage and weight of the part are minimized by using internal structures, which results in a cost and material-efficient production[27]. These internal structures include lattice structures, Voronoi Diagrams, channels, ducts and other complex geometries [28]. In the conventional machining processes, the part is shaped by the removal of the material, whereas in Material Extrusion type of AM systems, only the shell and internal structures are extruded through the nozzle. Also in powder and resin-based AM systems, the unprocessed regions of powders and resins can be used in the following productions.

2.2.4 Efficient Modelling and Representation Methods of Internal Structures in Additive Manufacturing

Since the applications of AM have widely spread, efficient modeling and representation become more important. Commonly utilized data formats in AM applications include STL, STEP, STEP-NC, AMF, and 3MF. The STL file format, a widely used data exchange format, encapsulates waterproof mesh data comprising vertices and faces of the specified CAD geometry. Similar to STL, the AMF format, which is XML-based, extends its utility by including additional details such as materials, colors, lattices, and constellations. The 3MF format, also rooted in XML, was developed in coordination with the Windows Operating System design, aiming to provide users with a seamless print control interface while addressing interoperability challenges. The STEP and STEP-NC formats facilitate data exchange, containing comprehensive details such as CAD geometry, tolerances, and specific AM process parameters. Notably, STEP-NC enables the physical device control of computerized numerical controllers (NC).

Other formats, such as IGES, NURBS, OBJ, and VRML, also offer varying levels of

capability, information fidelity, and accuracy for 3D Printing. However, these formats are less principal and infrequently employed among manufacturers and AM end-users [29]. Each data exchange format presents a unique set of advantages and disadvantages compared to others as detailed in Table 2.1.

Table 2.1: Advantages and Disadvantages of Data Formats in AM

Format	**Advantages**	**Disadvantages**
STL	Simplicity for processing, highly portable	No support for modern AM, error-prone, poor scalability
STEP	Supports precision mfg requirements	Computationally complex
STEP-NC	Supports precision mfg requirements	Paradigmally different, no tesselated model
AMF	Wide support of AM capabilities and future extensibility	Currently less widely adopted
3MF	Sophisticated process and metadata support for inter-operability	Currently less widely adopted

In their research, Yaman et al. [30] developed a new methodology to compress bitmap images used in fabrication with Digital Light Processing (DLP) type of 3D printers. The paper uses exclusive OR operations for relative encoding among consecutive binary images, which is uncommon in literature. By utilizing the developed methodology, the size of a binary image sequence can be significantly reduced compared to commercial CAM software and other compression methods. Moreover, another advantage of the proposed methodology is its ease of implementation and low computational requirements. This enables it to function without needing a host PC during fabrication.

In another study, a new method is proposed for efficient modeling and representation of the lattice structures, slicing algorithm in a streaming mode, density matched and self-supporting lattice structures considered. When parts get larger or the unit cell size of the lattice structure gets smaller, the modeling and representation of the lattice structures become challenging due to the excessive memory requirement. To

overcome these problems, the method includes an implicit solid representation of the lattice structure with a weighted graph where unit cells' radius values are stored in the edge weights. As stated in the article, implicit representation comes up with the advantages of easy computation of intersections, unions, differences, ease of handling of topological changes, and ease of testing of the points whether they are on the surface while having the disadvantages of difficulty of enumerating the points on the surface and describing the sharp features, slow rendering of the part [31].

Novel encoding strategies are developed by Vassier et al. [32] based on variations of the existing AMF data format and a new ad-hoc hybrid file format. The strategy mainly aims to encode the repetition of similar geometry fragments. The efficiency of the strategies is proved by encrypting repetitive lattice and support structures. The developed methodology also preserves the exactness of the geometry. It's possible to obtain data size reductions up to 84% compared to the size of files generated by the state-of-the-art approaches. The process of identifying repeated patterns is executed through a newly developed heuristic algorithm, specifically designed to address the complexities of the underlying Weighted Exact Cover problem.

In another study, researchers developed a novel method to reduce the data size of meshed 3D geometries featuring repetitive patterns. The key point in this method is to encode a repeated feature as a singular mesh entity, rather than encoding each similar mesh individually. This approach is particularly effective in structures with repeating patterns, such as lattice frameworks. Remarkably, this method has demonstrated a reduction in data size by up to 70% in certain meshes characterized by repetitive elements [33].

Within the DfAM domain, the CAD software nTopology employs a data transfer methodology that avoids the conventional meshing processes in cooperation with EOS. This method, known as *Implicit Interop*, is specifically engineered to overcome the challenges of AM techniques. *Implicit Interop* facilitates the seamless transfer of design data across platforms, including nTopology, EOS 3D Printing Solutions Company manufacturing machines, CAD, and CAE software, utilizing the nTop Implicit File (.implicit) format. This file format, as outlined in [34], provides advantages over the traditional BRep or mesh-based data formats, enhancing the efficiency and ef-

fectiveness of data handling in DfAM processes. It's stated that the following key attributes of the nTop Implicit File format underscore its superiority and efficiency:

- nTop Implicit Files can be up to 99% smaller than meshes.
- nTop Implicit Files are generated up to 500x faster and load in up to 60% less time in the receiving software.
- nTop Implicit Files are accurate and lossless geometry representations based on mathematical functions, not surface approximations.

2.2.5 Applications of Voronoi Diagrams

3D Voronoi tessellations have broad applications, encompassing mechanical engineering, civil engineering, architecture biomedical sciences, geosciences, and electronics.

Gomez et al. [35] used 3D Voronoi structures to mimic bone-like implantable structures (porous scaffolds), replicating natural bone properties across various levels including biological, microstructural, mechanical, and mass transport characteristics. The design process of 3D Voronoi structures is conducted using Rhino Grasshopper Software, with parts manufactured through Stereolithography to ensure biocompatibility. Mechanical and mass transport properties are modulated with gradient-controlled interconnected porosity, controlled pore distribution, and controlled pore shapes, benefiting from the unique properties of 3D Voronoi tessellation techniques.

In another research [36], initiative within the field of geosciences, 3D Voronoi tessellation and Delaunay triangulation are applied. While raster structures such as voxels and octrees are commonly used in the geographical information system (GIS) community, they exhibit limitations in modeling and analyzing 3D geoscientific fields. A novel method based on 3D Voronoi Diagrams and Delaunay tetrahedralization is proposed, offering significant advantages over the conventional tessellations. A key benefit of employing Voronoi Diagrams for modeling 3D geoscientific datasets is their ability to seamlessly handle faults and discontinuities in the data and phenomena.

Further research [37] in the electronics field explores the use of 3D Voronoi structures

for enhancing electromagnetic interference shielding in wearable pressure sensors. The sensor modeling is performed in Rhino 7 Grasshopper software. Various sensor types, with differing levels of porosity and porous structures tuned by the properties of 3D Voronoi Diagrams, are fabricated using FFF techniques. The proposed design approach demonstrates that intelligent sensors and wearable electronic devices can be designed and manufactured with reduced weight while improving electromagnetic shielding performance.

Another study [38] focuses on topology optimization using differentiable Voronoi Diagrams. The objective is to minimize mean compliance through this innovative method. The differentiable Voronoi Diagram offers an effective cellular representation for topology optimization, enabling a novel design space for effective exploration of design alternatives compared to conventional methods. The efficiency of the optimized cellular structure is demonstrated with examples of anisotropic cells, such as those found in femur bones and Odonata wings, which are cellular biomimetic structures.

An additional study [39] introduces a design methodology, employing 3D Voronoi Tube structures as structural grids for tall buildings. This approach investigates geometrical patterns, characterized by varying degrees of density and irregularity along the building's height, focusing on mechanical properties such as relative density, axial, and shear stiffness. Furthermore, optimizing member sizes is integrated into the process to fine-tune strength and stiffness throughout the building's elevation. The findings suggest that 3D Voronoi Tube structures are structurally efficient, concluding the high effectiveness of the proposed method.

The study of McMains et al. [40] focuses on the additive manufacturing of thin-walled structures in a faster way while decreasing material usage. The main point is offsetting the boundaries of 2D slices of thin-walled BRep geometry. While making this offset operation, Voronoi diagrams with signed distance functions are used. A height field is created by raising the vertices of Voronoi diagrams in z by their signed distance. Offsetting operation by n is the same as slicing the resulting Voronoi mountain with the plane $z = n$.

The Voronoi diagrams have their roots in nature and can be widely seen in many areas

in nature [4]. Due to their shape and complete tessellation of the plane, it's possible to use the area in an efficient way. Since all the space is partitioned, Voronoi diagrams give the possibility for squeezing as much as possible in a limited space, which can be considered a reason to be observed in nature. Moreover, if something is growing at a uniform rate from separate points, the Voronoi diagrams appear. Onion skin cells, muscle cross-section, garlic bulb, wings of a dragonfly, soap bubbles, leaves, giraffe coat patterns, corns, and jackfruit are examples of the Voronoi Diagrams observed in nature and illustrated in Figure 2.8.

Figure 2.8: Voronoi Diagrams in Nature [4]

CHAPTER 3

MODELING, COMPACT REPRESENTATION AND ADDITIVE MANUFACTURING OF 3D VORONOI BASED INTERNAL STRUCTURES

In this Chapter, a methodology for modeling, compact representation and AM of 3D Voronoi based internal structures are presented. The proposed method is employed on the *Rhinoceros 7* Software *Grasshopper* module.

Rhinoceros 7 is a Computer-Aided Design software that employs NURBS (Non-Uniform Rational Basis Splines) geometry. Widely utilized in architecture, prototyping, engineering, and jewelry design, the software offers robust modeling capabilities. Integrated within *Rhinoceros 7*, *Grasshopper* serves as a visual scripting module, empowering users to utilize built-in and custom scripts for modeling tasks. In the methodology, Grasshopper is used to implement the generative design approach and visualize it within the *Rhinoceros 7* canvas. Additionally, Grasshopper facilitates the export of the NC file for manufacturing and the developed custom data format **.ont**.

The application includes five main steps which are explained in the following sections of the Chapter.

- Pre-Processing of input parameters in Grasshopper
- Modeling of 3D Voronoi Structures using Voronoi3DGenerator Plug-In
- Visualization of output geometry
- Custom data format generation and application
- Manufacturing of output geometry

The overall Workflow of the whole process is given in Figure 3.1.

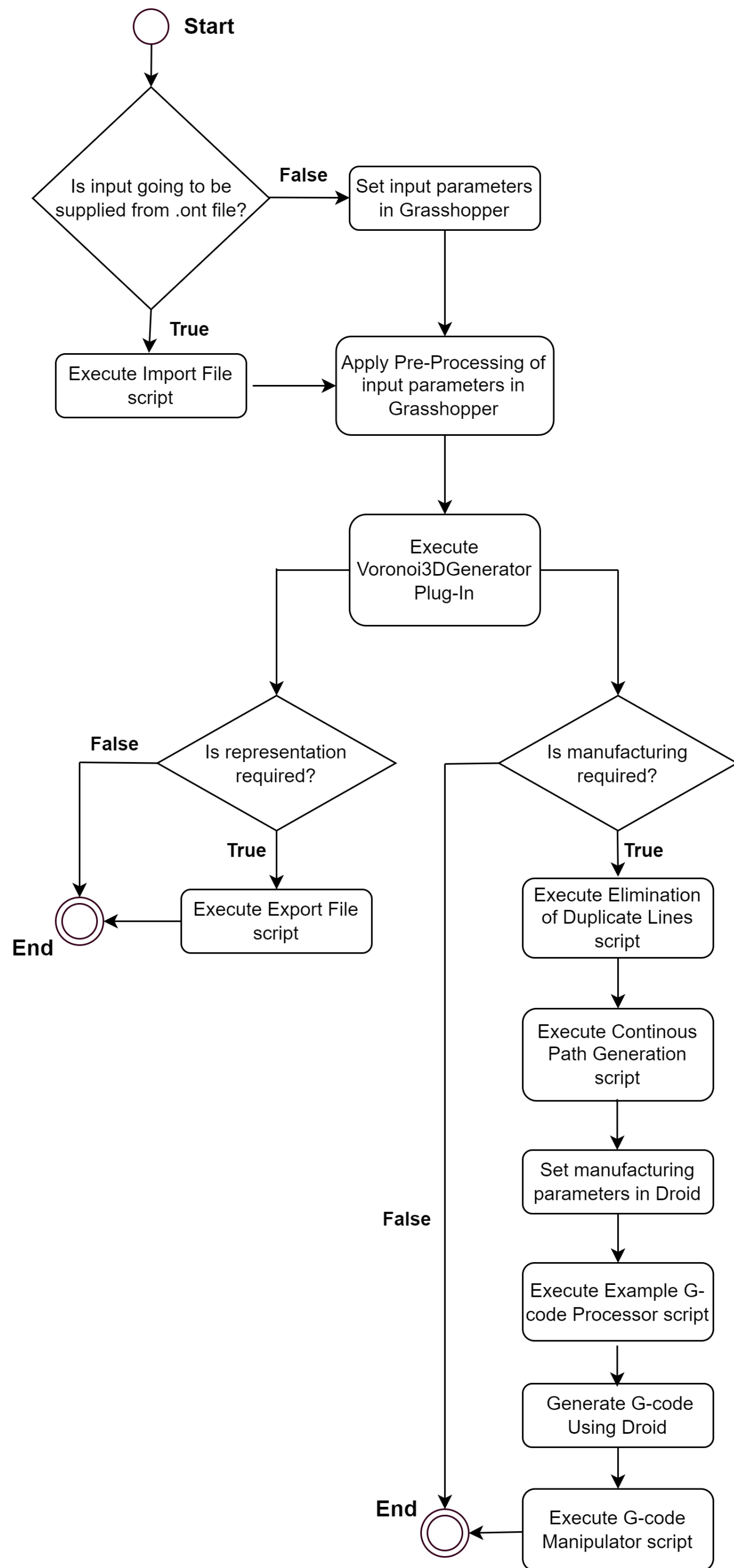


Figure 3.1: Overall Workflow of the Process

3.1 Input Parameters and Pre-Processing for Voronoi3DGenerator Plug-In

The input parameters set by the user within the Grasshopper interface for the proposed methodology, along with their detailed explanations, are as follows:

- Layer height lh *(double)*: Since the parts are manufactured layer by layer in AM, the thickness of the layer is an important parameter influencing the manufacturing process and quality of the final product.
- $offset$ *(double)*: Offset parameter is utilized to increase the weights of the cells in the Weighted Voronoi diagram generation. The use of parameter in application is explained in Section 3.10.
- Number of Voronoi Seeds *(integer)*: This parameter specifies the number of Voronoi seeds that will be inserted into the base layer for Voronoi generation. The use of parameters is explained in detail in Section 3.1.4.
- Mesh of the target part: The user imports the mesh in STL format into the *Rhinoceros* environment for subsequent processing.

This section includes the process of preparation of target geometry to be filled with Voronoi structures. The operation is done within *Grasshopper* by using the built-in blocks. The workflow is stated below in Figure 3.2.

3.1.1 Conversion from Mesh to BRep

Upon importing the STL formatted mesh file, it is rendered as a mesh within the *Rhinoceros Canvas*. The motivation for transitioning from Mesh to BRep is twofold:

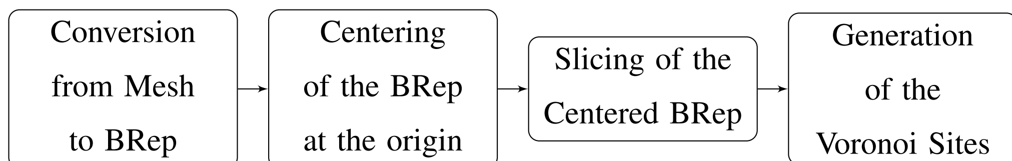


Figure 3.2: Grasshopper Workflow

firstly, it facilitates the utilization of component blocks in *Grasshopper* that are compatible with BRep input; secondly, the BRep structure affords a more streamlined and adaptable framework for geometric manipulations compared to its mesh counterpart.

Operational processes begin within the *Grasshopper Canvas*. The *Mesh* block is employed to retrieve the mesh from *Rhinoceros*, subsequently channeling it to the *Grasshopper* Canvas. Sequentially, the *FaceBoundaries* block is utilized to transform mesh faces into polylines. Thereafter, the *Boundary Surfaces* component is used to convert boundary edge curves into their respective boundary surfaces. Following this, the *Join* block unites all the created *Boundary Surfaces*. Following the process, the unified construct is collected within the *BRep* block. The schematic for the operation can be seen in Figure 3.3.

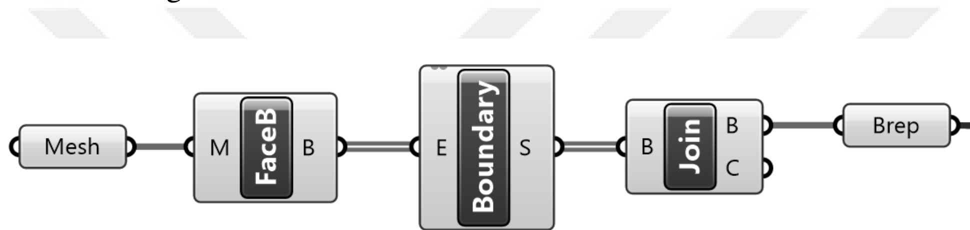


Figure 3.3: Mesh to BRep Conversion in *Grasshopper Canvas*

3.1.2 Centering of the BRep at the Cartesian Origin

The repositioning of the *BRep* such that it aligns with the absolute origin ensuring the BRep base remains 0.1mm beneath the WorldXY plane is crucial for the following application stages. This requirement arises due to the potential variability in the spatial location of the exported STL data within the global coordinate framework, which, in turn, could introduce complexities during the slicing and manufacturing phases. For the tasks described in this subsection, native *Grasshopper* modules are employed. The schematic can be seen in Figure 3.4.

The BRep input is taken from the output of the subsection "Conversion from Mesh to BRep" (Section 3.1.1). *Bounding Box* block is used to compute the minimum bounding box of the BRep object. Then, this bounding box is deconstructed by using *Deconstruct BRep* block to obtain its vertices. By using the *Vertices* output given as

a list, the vertices of the bounding box are collected. The first three vertex elements are positioned in the bottom of the box and these elements are collected by using *List Item* component by setting the indices as 0, 1 and 2. This vertices are used to create the bottom base rectangle by using *Rectangle 3Pt* block.

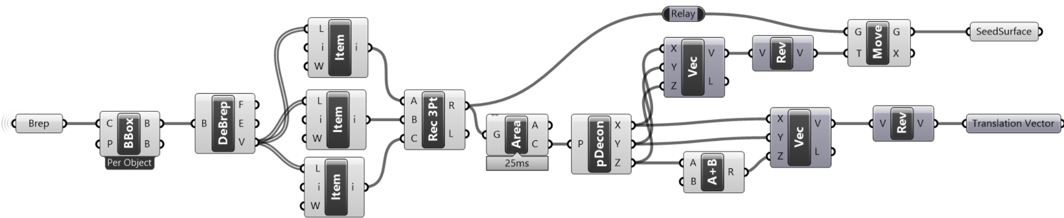


Figure 3.4: Schematic of Centering of the BRep at Cartesian Origin in Grasshopper

At this point, the flow goes in two different ways: obtaining the translation vector of the BRep and obtaining the surface where the Voronoi Seeds will be imported at WorldXY plane. The flow is very similar, the difference is that SeedSurface should be in WorldXY plane since the Voronoi Generation works efficiently in WorldXY plane. In other planes, some errors occurred during the generation of Voronoi structures. BRep should be 0.1 mm below the WorldXY plane to perform a slicing operation. To obtain the translation vector, the centroid of the bottom base rectangle is obtained by using *Area* block's *Centroid* output. This centroid point includes the X, Y, and Z coordinates of the translation vector to the Cartesian Origin. So, by using the centroid point's coordinates, a translation vector is constructed by using *Vector XYZ* block and reversed by using *Reverse* block. Also, since the BRep should be positioned below 0.1 mm of plane WorldXY *Addition* component used to add 0.1 mm to the Z coordinate of the *Translation Vector*. To obtain *SeedSurface*, the base rectangle is translated by not considering this 0.1mm addition.

The BRep input is derived from the product of the subsection titled "Conversion from Mesh to BRep" (see Section 3.1.1). Utilizing the *Bounding Box* block, the minimal bounding box surrounding the BRep object is computed. Subsequent to this, the *Deconstruct BRep* block facilitates the disassembly of this bounding box to extract its constituent vertices. Through the *Vertices* output, presented in list format, the bounding box's vertices are collected. Notably, the initial triad of vertex elements is

located at the base of the box. To acquire these foundational elements, the *List Item* component is invoked, designating the indices as 0, 1, and 2. These vertices are vital in the generation of the base rectangle through the *Rectangle 3Pt* block.

At this juncture, the procedural flow divides into two distinct paths: firstly, to derive the translation vector of the BRep, and secondly, to locate the plane onto which the Voronoi Seeds will be projected in alignment with the WorldXY Plane. Though these pathways exhibit marked similarity, it's important to highlight that for optimal Voronoi generation, the *SeedSurface* must conform to the WorldXY plane. Divergences from this plane have precipitated errors during Voronoi cell formation. Moreover, to implement the slicing operation, the BRep ought to be positioned 0.1 mm beneath the WorldXY Plane.

To extrapolate the requisite translation vector, the centroid of the base rectangle is identified through the *Area* block's *Centroid* output. This centroid inherently possesses the X, Y, and Z coordinates essential for translating to the Cartesian Origin. Consequently, with these centroid coordinates in hand, a translation vector is synthesized using the *Vector XYZ* block and subsequently inverted with the *Reverse* block. Given the requirement that the BRep maintain a position 0.1 mm below the WorldXY, the *Addition* component has been deployed to append this value to the Z coordinate of the *Translation Vector*. To extract the *SeedSurface*, the foundational rectangle undergoes a translation, omitting the previously mentioned 0.1mm adjustment.

The outcome of the translation operation is visually represented in Figure 3.5. This illustration showcases a Truncated Pyramid Mesh positioned precisely 0.1mm below the WorldXY Plane. Additionally, the terminal result of this translation procedure results in the construction of the base rectangle, denoted as *SeedSurface*.

3.1.3 Slicing of the Translated BRep

The BRep, constructed in Section 3.1.1, undergoes translation via the *Move* block, guided by the translation vector retrieved in Section 3.1.2. Post-translation, this translated entity is the input to the *Slice* block, an external command integrated within the *Xylinus* library. This specific block creates bounding curves, based on a user-specified

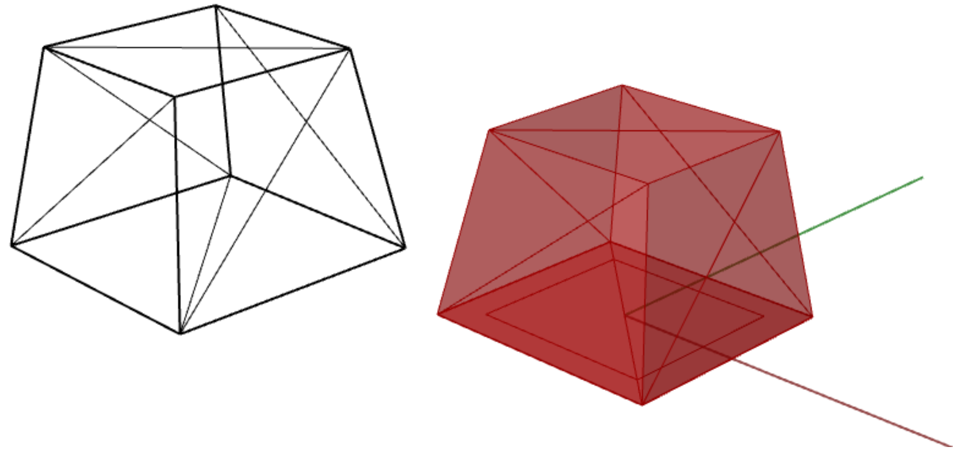


Figure 3.5: Centering Operation of Truncated Pyramid Mesh at the Cartesian Origin

layer height, represented as a double value. Subsequent to this, the *Slice* block's output is orthogonally projected onto the WorldXY plane utilizing the *Project* block. This operation is necessitated by the empirical observation that the Delaunay triangulation of *SuperDelaunay* Plug-In exhibits optimized performance when operating within the WorldXY plane. Alternative planes have been observed to cause complications.

To ensure fidelity in the *3D Voronoi Generator Algorithm*, it becomes essential to catalog the Z coordinates of the bounding curves, as these are required for accurate spatial transformation to their designated positions. To facilitate this, bounding boxes are synthesized around the bounding curves. Subsequently, these boxes are disassembled to their fundamental components via the *Deconstruct Box* block. The resultant output enumerates a domain of Z-coordinate values, typified by sequences such as "0 to 0" and "0.2 to 0.2". To further refine this data, the *Deconstruct Domain* block is employed, with its output undergoing a flattening procedure. This results in the acquisition of Z-coordinate values of the bounding curves arrayed as a list. Additionally, to fulfill requirements in the subsequent *3D Voronoi Generator Algorithm*, the layer count resulting from the *Slice* block is collected through the *List Length* block. A comprehensive representation of the workflow regarding the slicing operation is presented in Figure 3.6.

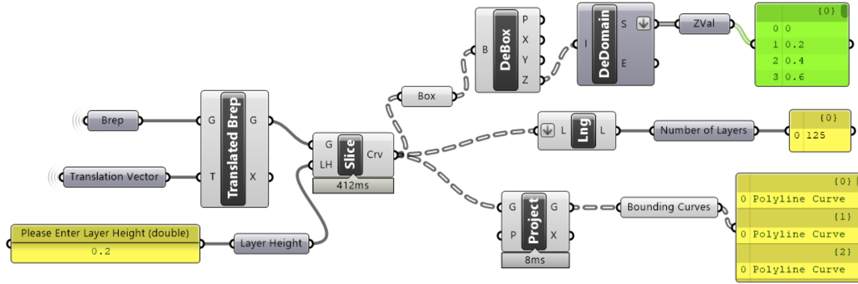


Figure 3.6: Slicing Operation of the Translated BRep in Grasshopper

3.1.4 Generation of Voronoi Seed Points

The generation of Voronoi seeds on the initial layer is performed by using *Populate 3D* block. The number of points inserted into the initial layer is set as an integer value. The surface for point insertion is taken from the output of Section 3.1.2 as *SeedSurface*. As mentioned in Section 3.1.2 the *SeedSurface* is derived from the bounding box of the BRep, so that the points will be inserted in the maximum cross-section of the BRep. At the end of the block, *SeedPoints* are obtained as *Point3D* objects. The Grasshopper schematic of the operation can be seen in Figure 3.7.

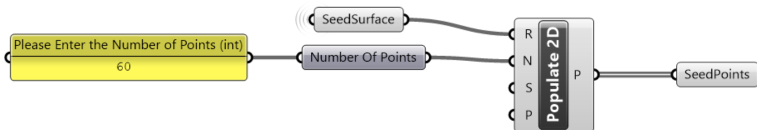


Figure 3.7: Initial Layer Seed Points Generation in Grasshopper

3.2 Modeling of 3D Voronoi Structures using Voronoi3D Generator Plug-In

A custom Plug-In is developed, named **Voronoi3DGenerator**, in Visual Studio 2019 using template **Grasshopper Assembly for Rhino 7 (C#)** for 3D Voronoi generation. The main algorithm is explained in Subsection 3.2.1 whereas the sub-algorithms are explained in the following subsections. The pseudocodes of algorithms used in the Plug-In are presented in **Appendix A**. For the simplicity of the notation, the symbolics

of the variables, listed in Table 3.1, are used in the explanations and pseudocodes of the algorithms.

Table 3.1: List of Variables

Name	**Structure**	**Type**	**Symbol**	**Description**
pointsList	List	*Point3D*	PL	Stores the Voronoi Seed points
insertedpointsList	List	*Point3D*	IPL	Stores the Voronoi Seed points
offsetList	List	*int*	OL	Stores the tag of cell offset
weightList	List	*double*	WL	Stores the weights for each point
sequenceList	List	*int*	S	Stores sequence of spanning tree
spanTopology	List	*int*	SL	Stores the minimum spanning tree topology
curveBounds	List	*Curve*	CB	Stores input boundary curves
densityList	List	*double*	DL	Stores Voronoi density at each layer
postoffsetList	List	*double*	POL	Stores the tag of cell offset in post offset
commonpointsList	List	*Point3d*	CP	Stores intersection points of three cells
targetpolyList	List	*Polyline*	TPL	Stores target Voronoi for post offset
voronoiTree	Tree	*Polyline*	VT	Stores Voronoi cells
cvoronoiTree	Tree	*Polyline*	CVT	Stores continuous Voronoi cells
neighTree	Tree	*Polyline*	NT	Stores neighbouring cell indices
mspanTree	Tree	*int*	MST	Stores SL considering continuous path
preoffsetTree	Tree	*Polyline*	POT	Stores voronoi cells before offset

3.2.1 3D Voronoi Generator Algorithm

Algorithm 1, **3D Voronoi Generator Algorithm**, is the main algorithm for generating Voronoi structures.

The algorithm requires the following inputs:

- List of points located in the WORLDXY plane which are the seeds of Voronoi cells (PL) result of Section 3.1.4
- Number of layers (LN) result of slicing operation explained in Section 3.1.3
- Z coordinates of the slices ($zslices$)
- ($offset$) value given as a user input for the offsetting Voronoi structures over the layers
- Outer bounds of the slices of BRep (CB)
- Layer height (lh) for FFF manufacturing set by the user

The algorithm produces the following outputs:

- The neighboring cell topology as a tree (NT)
- Inserted points list (IPL) for visualization of the points inserted
- preoffsetTree (POT) to visualize the cells before post-offset operation
- Minimum Spanning Tree (MST)
- Density list (DL) to observe the Voronoi density over each layer
- Resulting Voronoi tree (VT)
- Custom arrangement of (VT) for FFF manufacturing continuous Voronoi tree (CVT)

The workflow for the algorithm is illustrated in Figure 3.8

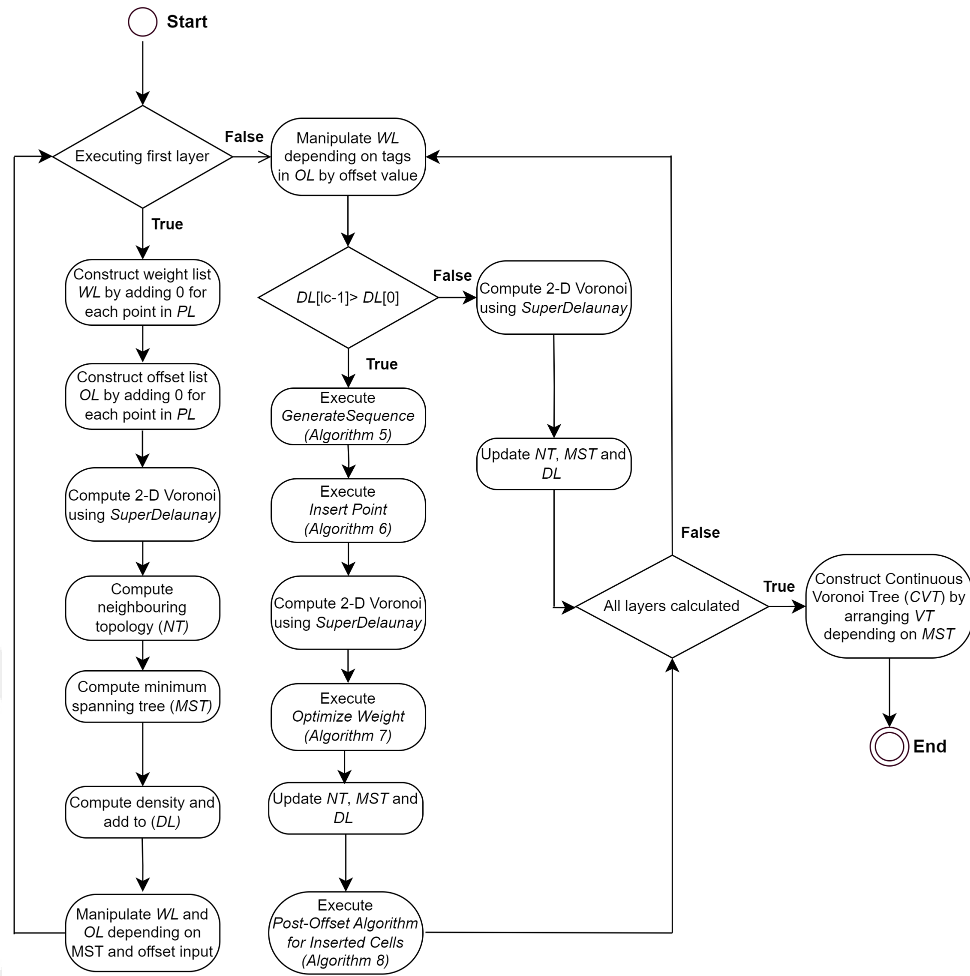


Figure 3.8: 3D Voronoi Generator Algorithm Workflow

The weight list WL includes the weights in the Power diagram for each point in PL, a key variable for offsetting Voronoi cells over the layers. Offset list OL is constructed to determine whether the cell is offset before over each layer. At the beginning of the algorithm, WL and OL are filled with zeros for each point in PL. Both WL and OL are updated over each layer (lc).

The input Curve Bounds CB are on WorldXY plane as stated in Section 3.1.3 since there existed problems while computing Delaunay triangulation other than WorldXY plane. Therefore the actual Z coordinates of the slices are stored in $zslices$ as a list to transform the computed Voronoi structures to their corresponding planes followed by (lc). The Weighted Voronoi diagrams are generated with an external Plug-In called *Super Delaunay*. The Delaunay triangulation and Voronoi generation are performed

for the seed points in PL constructed over the Bounding Box of the BRep as stated in Section 3.1.4. *Super Delaunay* generates Weighted Voronoi Diagrams with corresponding weight values of points and trims the geometry with curve bounds CB. The neighbouring topology (NT) and the minimum spanning tree (MST) are calculated using *Super Delaunay* depending on the generated Delaunay triangulation.

In a Weighted Voronoi Diagram (also known as a Power Diagram or Additively Weighted Voronoi Diagram), each site or seed point doesn't just have a location, but also a weight associated with it. The weight introduces a level of influence or power that each point has over its surrounding region as discussed in Section 2.1.2. The boundary between the two sites is influenced by both the distance to the sites and their weights. The boundary will be closer to the site with the smaller weight. Therefore, weight values are significant in expanding and contracting the Voronoi polylines. The Voronoi cells at the initial layer are constructed with zero weight values, making the first layer the same as the traditional Voronoi Diagram. Determining which points' weights will be increased at the first layer is done by considering the branches and edges of the calculated minimum spanning tree (MST). Elements at even levels (e.g., levels 0 and 2) are assigned a weight value corresponding to the offset provided as input within this spanning tree. In contrast, elements at odd levels (e.g., levels 1 and 3) were assigned a weight value of zero. As a result of this algorithmic approach, the expansion and contraction of Voronoi cells were not localized to specific regions but were uniformly distributed across the initial area. The calculated Minimum Spanning Tree MST is illustrated for the second layer of a truncated pyramid with 50 Voronoi seeds. The weights asserted to the odd-level cells are shown with circles in Figure 3.9.

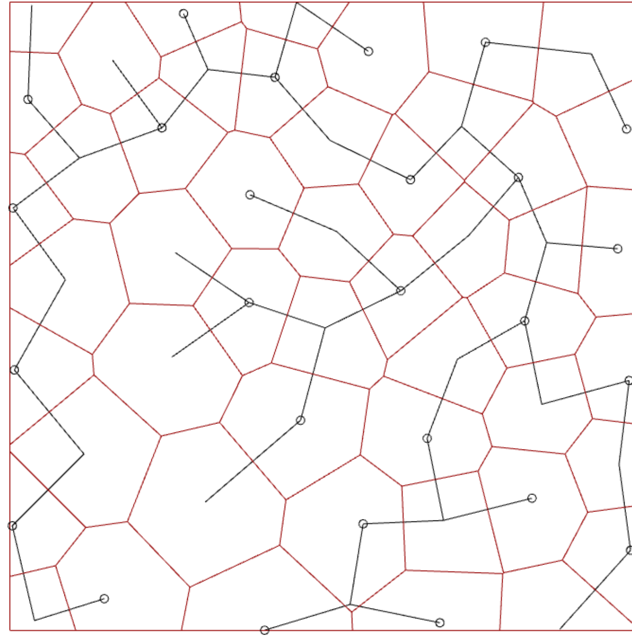


Figure 3.9: Minimum Spanning Tree Generation and Weight Assertion for a Truncated Pyramid

The preservation of convexity is considered significant due to the characteristics of Voronoi Diagrams. To ensure convexity across all layers, the $offset$ values added to weight of each cell are kept consistent. Consequently, when both neighboring cells underwent expansion, the shared boundary between them remained unchanged, then the convexity was conserved. Cells with a weight value of zero experienced gradual contraction and eventually disappeared over the following layers. As a result, a decrease in the number of Voronoi cells was observed in the following layers, leading to the progressive enlargement of the expanding Voronoi cells.

For each layer, the area of curve bounds $CB[lc]$ was calculated and the number of cells appearing in the corresponding layer was collected. A density value defined as the ratio of the area to the number of polylines is calculated and added to the density list DL. If the current layer's density is above the initial layer density then **Voronoi Structure Generation and Point Insertion Algorithm** (Algorithm 6) is called and one seed point is inserted with the weight value of cells getting larger. A Voronoi cell created with an inserted point is observed to be of substantial size, causing challenges in AM due to the absence of a supporting curve beneath, which could result

in overhangs. Therefore, the minimum possible weight value for the inserted point is calculated and the Voronoi diagram is constructed again with Algorithm 7, **Optimization of Weights in Voronoi Structure Generation (OptimizeWeight)**. For further reduction of the size, **Post-Offset Algorithm for Inserted Cells**, Algorithm 8, is applied for the inserted points to achieve the minimum possible cell size without changing the number of vertices before offset operation. The pre-offset Voronoi tree, before post-offset operation, POT, is also collected for visualization and debugging purposes.

The resulting Voronoi Tree structures for each layer are re-arranged and stored in continuous Voronoi tree CVT by considering the MST for the maximum continuous material extrusion in FFF production. The cells in branches of VT are re-ordered considering the indices of cells in the branches of MST and CVT are constructed.

Consequently, neighbouring topology treeNT, inserted points list IPL, pre-offset Voronoi tree POT, minimum spanning tree MST, density list DL, Voronoi tree VT, and continuous Voronoi tree CVT and loft tree LT are obtained as the outputs of the algorithm.

A Plug-In named **Voronoi3DGenerator** including the proposed **3D Voronoi Generator Algorithm** is developed in Visual Studio C# Grasshopper Template. The corresponding Grasshopper block can be seen in Figure 3.10.

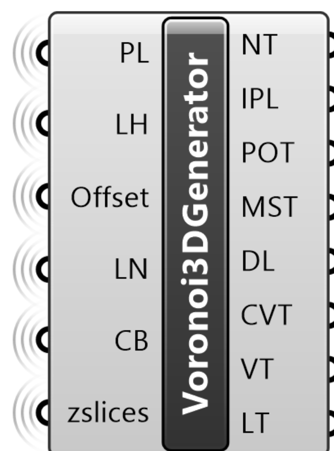


Figure 3.10: Voronoi3DGenerator Plug-In

An implication of the Plug-In is demonstrated in Figure 3.11 and Figure 3.12 across layer 60 and layer 190 for truncated pyramid geometry. Offset is 0.1, the layer height is 0.2, and the number of initial Voronoi seeds is 50. The inserted points are shown with their indices. As shown in Figure 3.13, the density values are above the initial density until layer 8 due to diminishing cells. Therefore, points are inserted until layer 7 to decrease the density. After layer 7, since the geometry is a shrinking truncated pyramid, density values continue decreasing and there is no need for an additional point insertion. The boundary curve shrinks over the layers and the contracting and expanding cells can be seen. Moreover, some of the cells are diminished due to the changing boundary curve and size of the weight of the neighbouring cells.

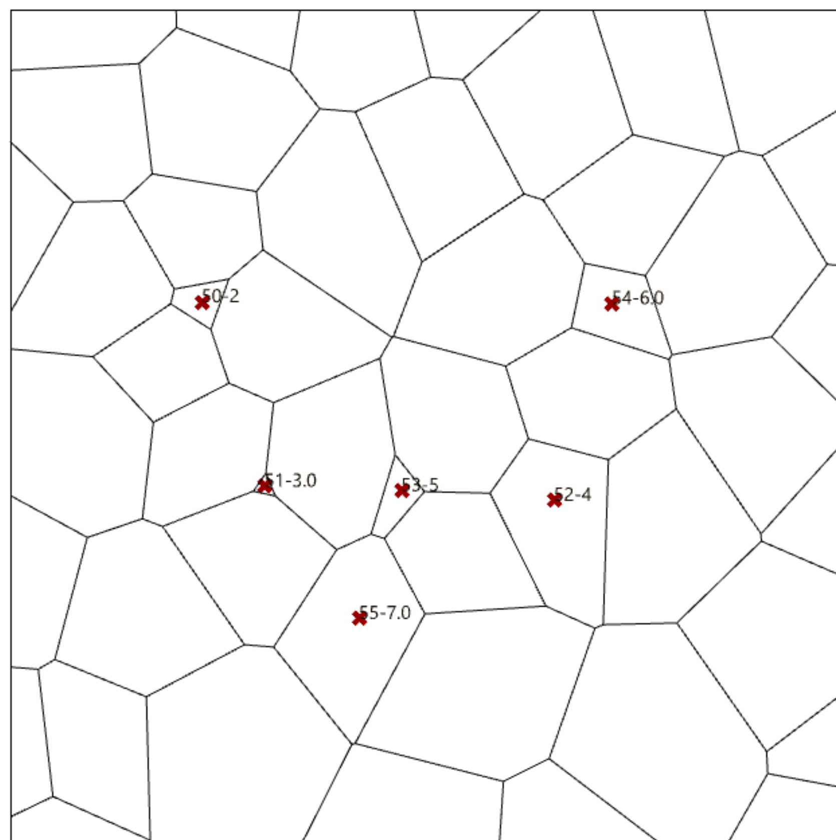


Figure 3.11: **Voronoi3DGenerator** Result for Layer 60

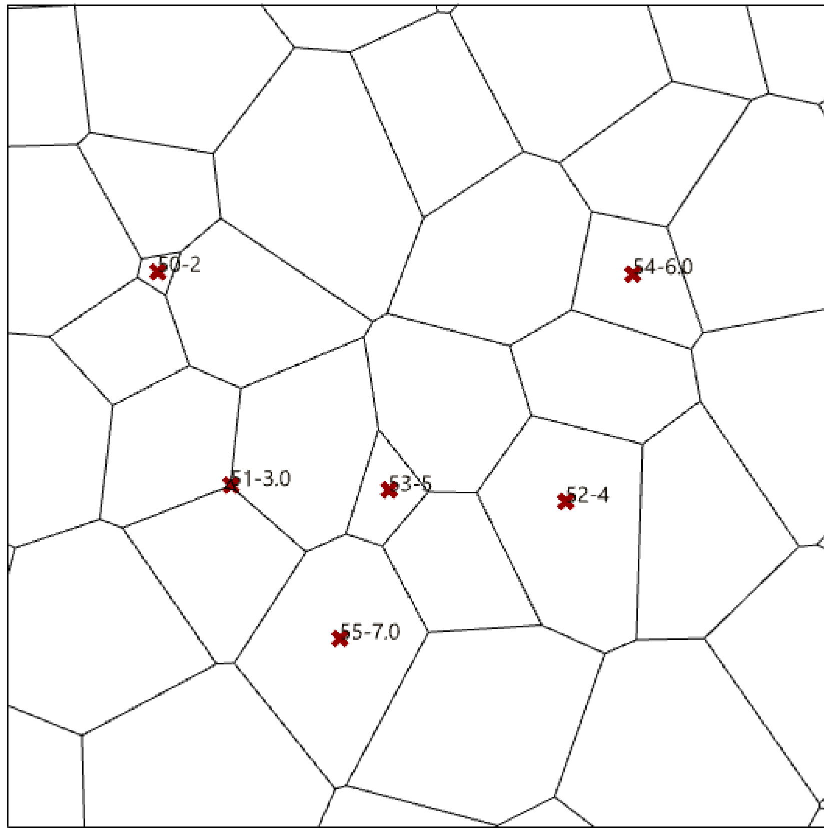


Figure 3.12: **Voronoi3DGenerator** Result for Layer 190

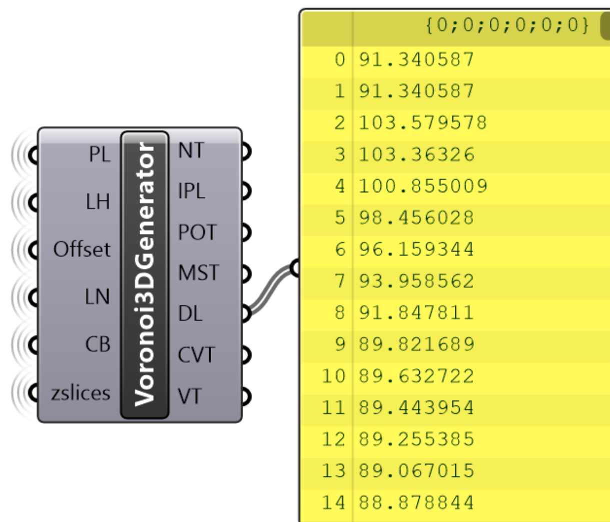


Figure 3.13: Results of *Density List (DL) for the Example Application*

3.2.2 Proximity Check Between Polyline Points and a Curve Algorithm

The pseudocode outlined in Algorithm 2, **Proximity Check Between Polyline Points and a Curve**, performs a proximity check between a polyline and a curve. It determines whether any point of the polyline, denoted as $\mathcal{P}$, is within a specified tolerance distance from the curve, represented as C. The inputs are the polyline $\mathcal{P}$, the curve C, and a tolerance value τ. The algorithm iterates over each point P_i in $\mathcal{P}$, computing the closest point C_{closest} on C for each P_i. It then calculates the distance d between P_i and C_{closest}, checking if d is less than τ. If $d < \tau$ for any P_i, the function sets a Boolean variable a as true and terminates, indicating proximity between $\mathcal{P}$ and C. If no point in $\mathcal{P}$ is within τ from C, a remains false, signifying that the polyline and curve are not proximate based on the given tolerance. This function is used in the program to control whether the edges of a Voronoi Cell are on the boundary curve CB or not.

3.2.3 Precision Adjustment of 3D Point Coordinates Algorithm

Algorithm 3, **Precision Adjustment of 3D Point Coordinates**, focuses on the precision adjustment of a three-dimensional point's coordinates. Given a 3D point P and a specified number of decimal places n, the function's goal is to create a new 3D point P' whose coordinates are rounded versions of the original point's coordinates. The resultant point P' has its coordinates finely tuned to the required decimal precision, making this function particularly useful in the program where precise geometric positioning and calculations are crucial.

3.2.4 Distinct Point Enumeration in a Geometric Polyline Algorithm

Algorithm 4, **Distinct Point Enumeration in a Geometric Polyline**, is proposed to count unique points in a geometric polyline, symbolized as $\mathcal{P}$. It creates an empty HashSet S to store distinct points. Iteratively examining each point P_i in $\mathcal{P}$, the algorithm adds P_i to S only if it's not already present, ensuring the uniqueness of points. The outcome is the count of unique points in S, representing the total number of distinct points in $\mathcal{P}$, thus offering the elimination of the line segments whose two

endpoints are at the same location with zero length.

3.2.5 Symmetric Index Sequence Generation from a Voronoi Structure Algorithm

Algorithm 5, **Symmetric Index Sequence Generation from a Voronoi Structure (GenerateSequence)**, aims to create a symmetric sequence of indices from a Voronoi structure V, generating a list S. It starts by initializing S and calculating the total cell count N in V. For each cell index i up to $\frac{N}{2}$, it adds both i and its symmetric counterpart $N-1-i$ to S. If N is odd, the middle index $\frac{N}{2}$ is also included to maintain symmetry. The outcome is S, representing a symmetrically ordered index sequence from V. This algorithm plays a crucial role in ensuring a homogeneous distribution of inserted points across different layers. It achieves this by symmetrically altering the input of the spanning tree list. Such a modification effectively prevents the clustering of inserted point locations along the same branch of the spanning tree, thereby providing a more uniform distribution.

3.2.6 Point Insertion Algorithm

Algorithm 6, **Point Insertion Algorithm (InsertPoint)**, aims to update the Voronoi structure VT by inserting new points based on a predefined sequence list S, while considering a boundary curve CB and a list of density values DL. It analyzes each layer beyond the initial one, assessing the layer's density against the initial layer's to decide whether new point insertions are necessary for maintaining constant density over the layers. **Point Insertion Algorithm (InsertPoint)** aims to locate the inserted point at the intersection of three neighboring cells.

A particular cell $\mathcal{P}$, which is expanding (indicated by an offset value of 1 in OL) and not situated on the boundary, is chosen as the target cell for point insertion. The algorithm then explores the neighboring cells $\mathcal{P}_n$ and $\mathcal{P}_o$, filtered by specific criteria: they should not be on the boundary, should not be offset, and must not be recently inserted cells. When these conditions are met, the intersection point I of these cells is determined. If suitable neighbors are not found, the search continues with other

potential cells. The layer does not undergo point insertion without an appropriate $\mathcal{P}$.

Upon identifying the intersection point I, it is added to the PL for subsequent layer generation. Additionally, a weight w, calculated as $(lc - 1) \times$ offset, is assigned to ensure that I's influence grows in the following layers. Concurrently, the algorithm updates the weight and offset status of the target cell $\mathcal{P}$ in OL and WL to prevent further expansion in subsequent layers. This algorithm aims to obtain a balanced and strategically constructed Voronoi structure.

An application of the **Point Insertion Algorithm** is presented in Figure 3.14 and Figure 3.15 between layer 2 and layer 3 for truncated pyramid geometry. The input parameters are set as Offset = 0.1, layer height = 0.2, and number of initial Voronoi seeds is 50. The Voronoi cells are shown with their indices. Since the density increases at layer 2, a point with an index of 50 is inserted. It can be seen that the inserted point is located at the intersection point of the cells with indices 19, 35, and 49.

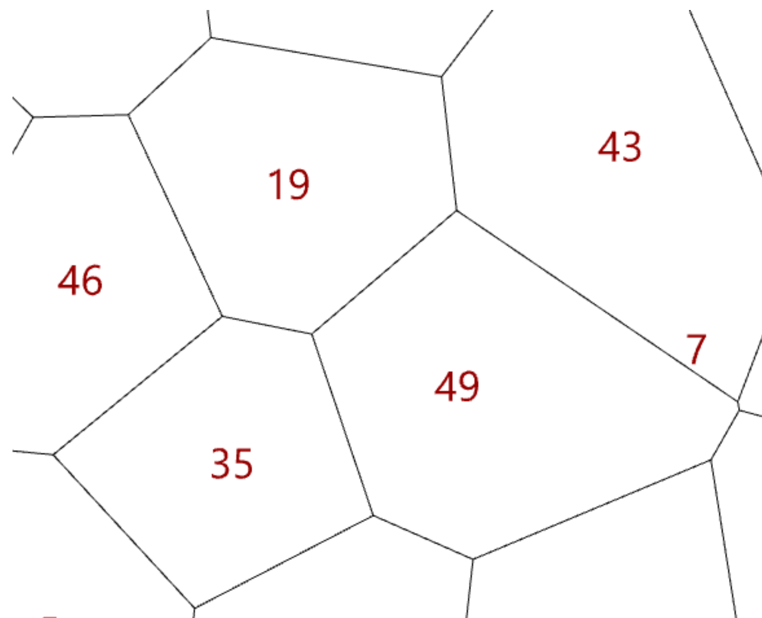


Figure 3.14: Visualization of Target Layer Before Point Insertion

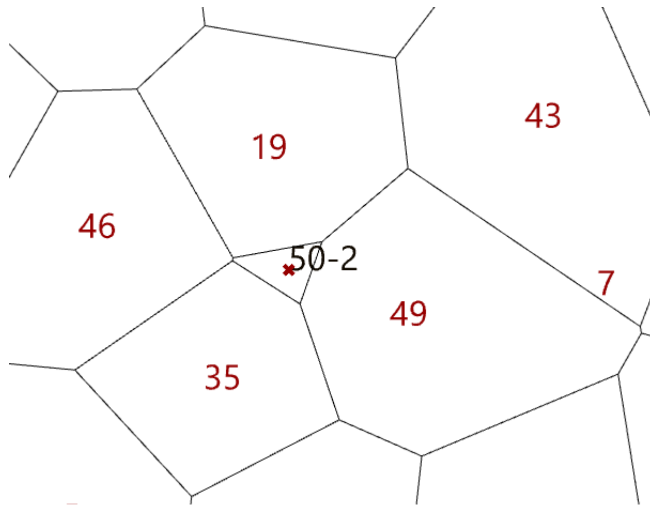


Figure 3.15: Visualization of Target Layer After Point Insertion

3.2.7 Optimization of Weights in Voronoi Structure Generation Algorithm

Algorithm 7, **Optimization of Weights in Voronoi Structure Generation Algorithm**, targets optimizing the weights (WL) of the inserted points in inserted point list (IPL) in a Voronoi structure (VT) to ensure cell formation with the possible minimum weight. For each point (P_i) inserted to VT beyond a specific layer, the algorithm adjusts its weight (WL_i) to maintain the existence and appropriate size of its corresponding Voronoi cell (VC). Initially, it reduces (WL_i) iteratively until VC ceases to exist, then slightly increases (WL_i) to find the minimum weight necessary for VC's existence. In subsequent layers, if VC is found to have more vertices than when it was first inserted, the algorithm further reduces (WL_i) to decrease VC's size. This is done while recalculating the Delaunay and Voronoi structures after each weight adjustment to ensure the integrity of VT over the layer. This process aims to enhance the manufacturability of the Voronoi structure in material extrusion-based systems through the optimization of the weights. This optimization is particularly critical as it directly influences the size of the inserted points corresponding Voronoi Cells (VC). A smaller VC reduces bridging distance, eliminating the failure risk in the Material Extrusion type of AM.

Moreover, the algorithm conducts a comparison between the unique number of points

of the inserted Voronoi Cell (VC) in its initial layer of insertion and the current layer. This constancy is crucial since the variation in the number of unique vertices leads to discontinuities in 3D Voronoi Cells across the layers. If the inserted Voronoi Cell's number of unique vertices is larger than the number of unique vertices at the initial insertion layer, the offset tag (OL_i) is set to be -1 to decrease the weight of the inserted cell in the following layers. This procedure is only applied to the inserted Voronoi cells.

3.2.8 Post-Offset Algorithm for Inserted Cells

Algorithm 8, **Post-Offset Algorithm for Inserted Cells**, is designed to refine a Voronoi structure, denoted as V, through the adjustment of vertices and optimization of the structure. This process is guided by taking inputs of VT, PL , CB, OL, WL, NT. The underlying idea for this refinement is to minimize the bridge lengths of inserted Voronoi cells, as the edges of these cells are situated on the edges of adjacent cells. Excessively elongated edges can cause significant manufacturing challenges. Therefore, the main objective of the algorithm is to achieve maximal refinement of the cells by ensuring that the number of vertices in the refined cell ($uniPoly$) is the same as that of the non-offset cell, thereby optimizing the structure for manufacturability.

The algorithm's core iteratively processes each layer lc within the Voronoi structure VT, focusing on inserted points. For each point under consideration, the algorithm first ensures that the corresponding Voronoi cell exists. If such a cell is present, the target cell is added to the target polyline list TPL for further operations.

When a target point is processed for the first time, the algorithm triggers a sequence of offset operations. The goal is to perform the maximum offset while maintaining the number of vertices in the refined cell equal to that in the non-offset cell. This is achieved using the offset functions the *Clipper* Plug-In provides. The offset operations are performed in WorldXY plane, then it's transformed to the corresponding plane with proper Z coordinate due to the requirements of the Plug-In. To ensure smooth fabrication, the area of the resulting polyline is controlled in addition to maintaining the number of vertices. During post-offset, it's ensured that the area of the

resulting cell does not go below $2mm^2$.

For points previously processed in earlier layers, the offset value is determined based on the layer where the point was initially processed. Additionally, the algorithm checks if the Voronoi cell ceases to exist following the offset operation. If so, the offset value is added to post-offset list POL to ensure the existence of the Voronoi cell.

The offset polyline is then collected, but the process does not conclude there. As a following step, the algorithm translates the edges of neighboring cells to the nearest points on the offset polyline, thus maintaining continuity. An additional algorithm is applied to identify the closest point, determining which adjacent cell vertex aligns with each vertex of the offset polyline. A further check ensures that vertices intended for transformation are not located on the Boundary Curve (CB). This is critical as transforming boundary curve vertices would break the integrity of CB. Therefore, these boundary vertices are preserved unchanged and flagged. Additionally, vertices of the offset polyline (Pex) corresponding to these boundary vertices are marked as excluded, as they should not be present in the final geometry.

Following the translation of adjacent polyline vertices, the next step involves updating the target polyline within the structure. This is accomplished by computing a convex hull encompassing the offset polyline's vertices (excluding Pex) and the boundary vertices. Subsequently, the Voronoi structure VT is updated with the geometry derived from this recently calculated convex hull.

In Figure 3.16, the output of the algorithm is illustrated. The red cell shows the cell before post-offset operation, and the cell colored black shows the resulting cell after post-offset operation. The operation is performed for the inserted cell with index 50 and the cell is inserted at layer 2. After the post-offset operation, the number of vertices in the target cell remains unchanged at 4, as shown in the figure.

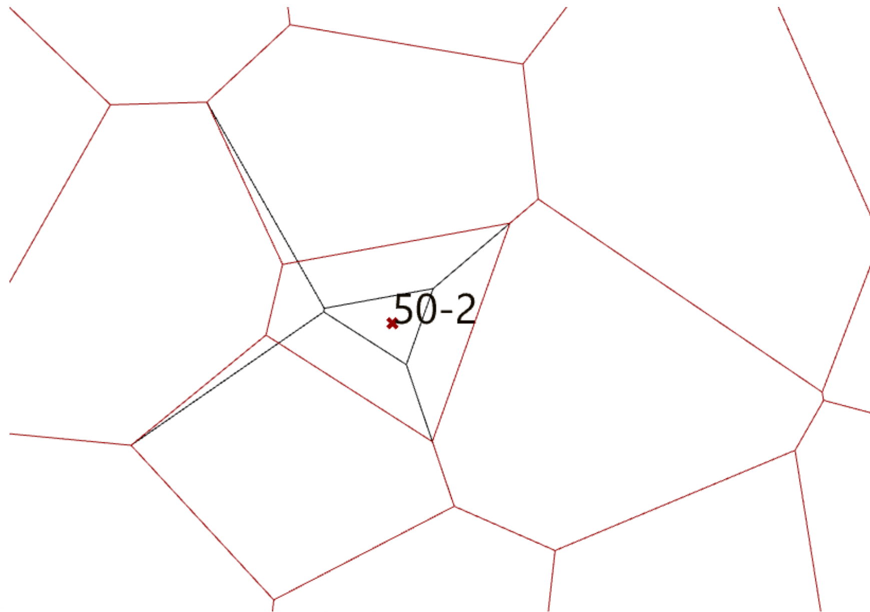


Figure 3.16: Visualization of Post-Offset Operation

3.3 Visualization of Output Geometry

The visualization of corresponding outputs of **3D Voronoi Generator** Plug-In is performed in Grasshopper canvas, by using Stream Gate blocks in Grasshopper to select the target visualization geometry. Preview Gate Block and the target geometries can be seen in Figure 3.17.

Figure 3.17: Preview Gate Block

Target geometries include:

- Full Geometry: The whole Voronoi structure is previewed.
- Target Layer: Only the target layer is previewed.

- Target Cell: Only the target cell at the target layer is previewed.
- Target Cell Neighbours: Target cell's all neighbours are previewed.
- Target Cell's Specific Neighbour: Target cell's neighbour with a specific index is previewed.
- First layer's Minimum Spanning Tree: The Minimum Spanning Tree at the first layer is previewed.

The previewing options are beneficial since it's very challenging to visualize specific sections of the complex geometries. The output geometry appears to be Voronoi structures as lines layer by layer and these layers can be very close to each other such as 0.1 mm, therefore, it's very difficult to distinguish individual lines. By using the preview interface shown in Figure 3.18, it's possible to specify and visualize the specified geometry depending on user inputs from *Panel* and *Number Slider* blocks. This feature is very crucial, especially in the development and test stages of the Plug-In, where effective examination of the geometry is required.

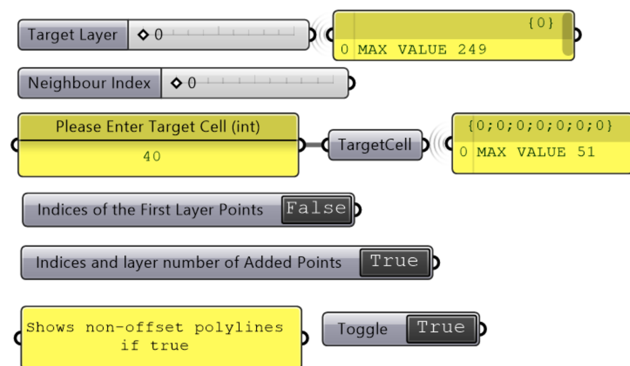


Figure 3.18: Preview Interface

3.4 Custom Data Format Generation and Application

The provided pseudocode of Algorithm 9 titled **Exporting .ont Data Format**, outlines an algorithm for processing input of Plug-In component to export the suitable special file format called **.ont** for compact representation and data size reduction.

The **.ont** file is in the form of a standard .txt file where the input variables are exported as text. At the beginning of the file, the variables offset value ($offsetValue$), layer height (lh) and seed coordinates list (PL) are written into a text file. Then, the boundary mesh (STL input geometry) is triangulated and converted to the ASCII format suitable for writing into a .txt file in the specified path ($outputPath$).

The Grasshopper C# block for the proposed algorithm can be seen in Figure 3.19.

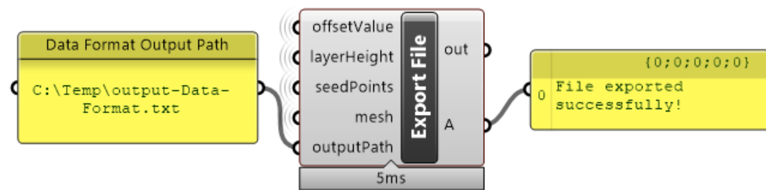


Figure 3.19: Export File C# Block

The structure of **.ont** data representation is illustrated in Figure 3.20.

```
[OFFSET_VALUE]
0.1   //Example double value for offset

[LAYER_HEIGHT]
0.15   //Example double value for layer height

[INITIAL_LAYER_VORONOI_SEED_COORDINATES]
X,Y,Z
1.0,2.0,3.0
4.0,5.0,6.0
7.0,8.0,9.0
...

[BREP_PART_STL]
solid part_name
  facet normal ni nj nk
    outer loop
      vertex v1x v1y v1z
      vertex v2x v2y v2z
      vertex v3x v3y v3z
    endloop
  endfacet
  ...
endsolid part_name
```

Figure 3.20: Output Data Format Structure

Algorithm 10, titled **Importing .ont Data Format**, is designed to read and interpret data from **.ont** file, reconstructing a 3D geometry and extracting relevant parameters like the variables $offsetValue$, lh and seed coordinates list PL. Initially, the algorithm checks for the validity of the provided file path and proceeds to read the file line by line. It parses different sections marked by specific keywords: "[OFFSET_VALUE]", "[LAYER_HEIGHT]", and "[VORONOI_SEED_COORDINATES]". For each section, it extracts and stores corresponding $offsetValue$ and lh as double, and seed coordinates list (PL) as Point3D objects. Furthermore, the algorithm processes mesh data located between "solid" and "endsolid", constructing a 3D mesh from the parsed vertex data. The final output includes $offsetValue$, lh, seed coordinates list PL, and the reconstructed mesh which will be processed to give input to Plug-In as input parameters.

The Grasshopper C# block for the proposed algorithm can be seen in Figure 3.21.

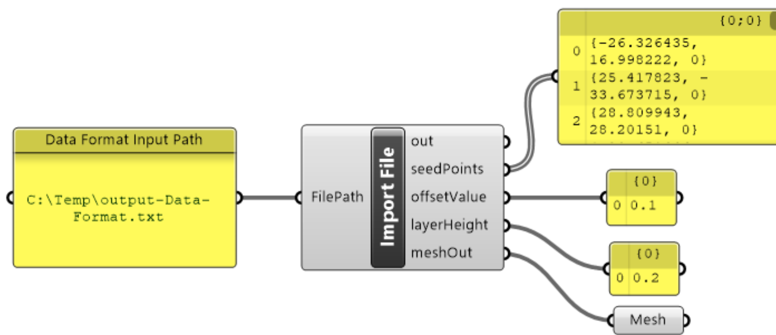


Figure 3.21: Import File C# Block

3.5 Manufacturing of Output Geometry

The method employed to fabricate the target geometry is explained in this section. Due to the nature of the proposed methods, there existed duplicated lines where two adjacent cells share in VT. Firstly, these lines are eliminated. Then an NC file is generated with external Plug-In *Droid* with the desired manufacturing parameters for FFF. Since *Droid* does not include machine-specific commands and parameters, the generated NC file is manipulated by using an example NC file generated in the

suggested slicing software of the 3D printer.

3.5.1 Elimination of Duplicate Polylines

The output geometry CVT is a data tree whose branches include Voronoi cells as polylines in corresponding layers. Due to the Voronoi 3D Generator Plug-In's nature, duplicate lines exist since two adjacent cells share the common edge in the same layer. When the manufacturability is considered, these common lines cause the nozzle to extrude material to a line segment twice in a layer, then the target layer height cannot be obtained. Therefore, these common lines are eliminated by using the Algorithm 11, **Eliminate Duplicate Lines in Voronoi Structure**. The algorithm focuses on analyzing and processing a data tree of Voronoi polylines, symbolized as CVT. Its primary objective is to traverse each branch within CVT, converting polylines into NURBS curves and further decomposing them into their constituent curve segments. Unique curve segments are identified and collected, ensuring no duplicates are retained. This collection of unique curves is then assembled into a new data tree, as $resultTree$ ready for NC file generation.

3.5.2 Continous Path Generation for Fabrication

The generation of the continuous path is crucial for extrusion-based additive manufacturing systems. Discontinuous paths necessitate the retraction of the filament, a mechanism that not only burdens the machinery but also poses a risk of residual material accumulation. Therefore an algorithm written in Grasshopper C# script has been found [41] and implemented. This algorithm, instrumental in processing the output from Section 11, efficiently handles a set of curves across various layers. The operational principle involves initiating from a user-defined curve and sequentially navigating to the nearest unvisited edge within the same layer. Encountering a previously traversed curve triggers an upward movement to a user-specified height, a lateral transition to the coordinates of the preceding line, and a subsequent descent. This procedure is reiterated until all curves in the layers are sequentially covered.

While the algorithm does not ensure an entirely continuous path, its capability to

maintain edge-bound movement is considerably advantageous for the fabrication process, a topic further elaborated in Section 3.5.4. An example path generated for a set of curves within the same layer is represented in Figure 3.22.

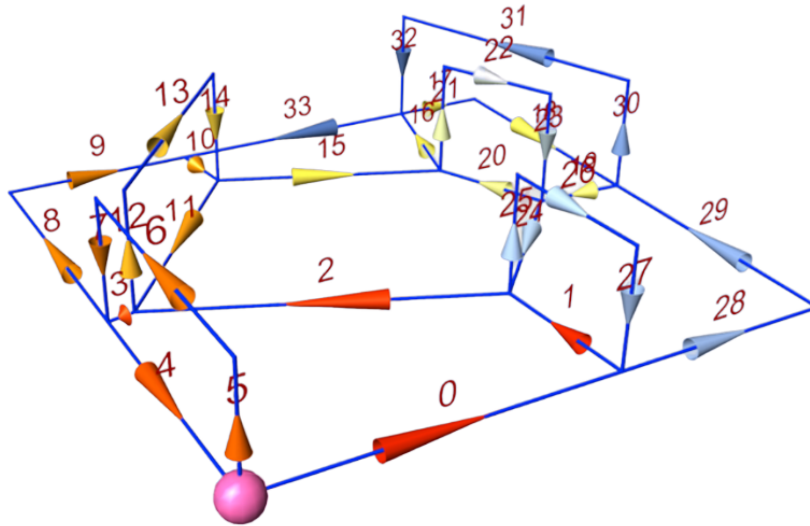


Figure 3.22: Generated Path Using Continuous Path Generation Algorithm

3.5.3 NC File Generation for FFF Systems

For the manufacturing of output geometry, FFF systems are preferred since the output geometry includes lines. FFF machines can easily manufacture these lines when the nozzle follows the line segments. When the diameter of the nozzle and extrusion width get larger, it's possible to obtain thinner wall structures. By using larger diameter nozzles, it's possible to produce geometries with higher offsets since there will be material below from the previous layer.

A Plug-In called *Droid* is used to generate an NC file from line segments by setting the input parameters presented in Table 3.2.

Each FFF machine may have different NC file commands for the desired operations such as heating the bed, nozzle, setting z offset, etc. On the other hand, *Droid* Plug-In does not have an FFF machine database and is not familiar with the target machine's NC file commands. Therefore, an example NC file prepared for the target FFF machine is required for the operations before and after printing starts. An ex-

Table 3.2: Fabrication Parameters for 3D Printing

Fabrication Parameters	**Value**
Layer Height (mm)	0.2
First Layer Height (mm)	0.2
Nozzle Diameter (mm)	0.8
Infill	0
Print Speed (mm/s)	10
Travel Speed (mm/s)	50
Retraction (mm)	0
Filament Diameter (mm)	1.75
Flow Rate Percentage Multiplier	100

ample NC file including these commands can be obtained from the common slicing software (e.g. Cura) with a machine database. The Algorithm 12, titled **Example NC File Segregation Based on Line Markers**, is used to collect these commands from an example NC file in .txt format. The user manipulates the .txt file manually, by adding markers "START" before the line where the printing begins and "END" after the lines where printing finishes. The algorithm processes the text file, specified by its path ($filePath$). Its main function is to categorize the file's content into two segments: $prependedText$ including Header and $appendedText$ including Footer. The algorithm identifies key markers within the file, labeled "START" and "END". It scans each line by order and adds the lines before the "START" marker to the $prependedText$ list and those following the "END" marker to the $appendedText$ segment.

In the output of Section 3.22, continuous path curves are moved to the center of the build plate, since Droid generates the NC file related to the build plate origin located in the bottom left (absolute origin) of the build plate.

The manufacturing parameters in Table 3.2, continuous path curves, $prependedText$ including Header and $appendedText$ including Footer are given as inputs to *NC File Generator Block*. The generated NC file is saved in .txt format to the specified path and printing information is presented in the panel. The described block is illustrated

in Figure 3.24.

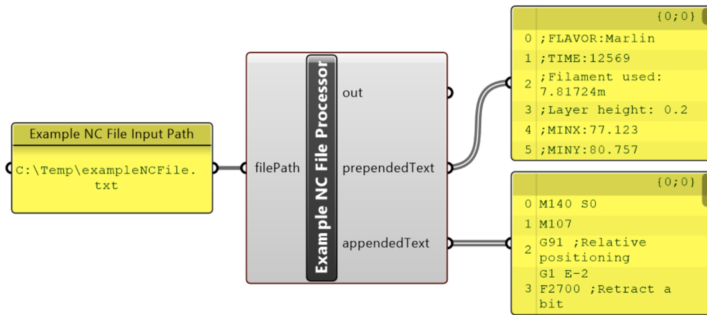


Figure 3.23: Example NC File Processor Block

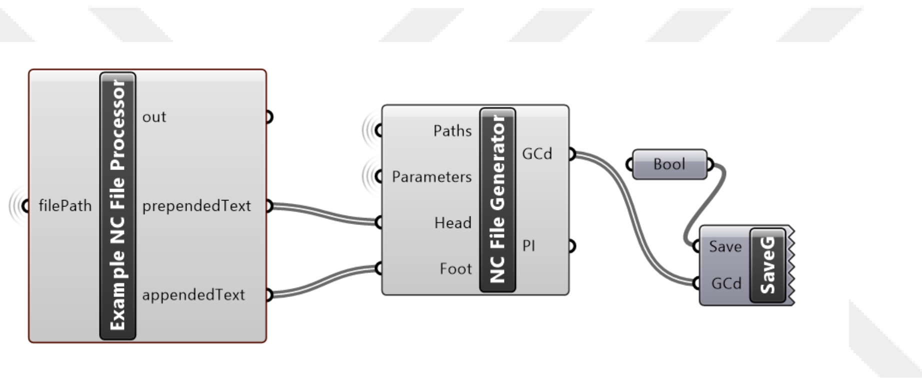


Figure 3.24: NC File Generator Block

3.5.4 NC File Manipulation

When the continuous path generated in Section 3.5.2 is converted to an NC file, as explained in Section 3.5.3, and imported into *Cura*, the nozzle typically follows the path along the Voronoi edges. Therefore, if there occurs a small amount of residual material on this non-extrusion movement, it will be mostly on the Voronoi edges and in the following layers, it will be compensated. However, while making the fast travel movements, it's observed that the nozzle extrudes material, which should be solved to obtain a successful fabrication. Also, even if this extrusion is eliminated, it's observed that when the nozzle makes the fast travel at the user-specified height, the filament flows outside the nozzle due to a sudden pressure drop. Therefore, the

generated NC file should be manipulated. The simulation in *Cura* for the first layer of manufacturing Cone geometry before the manipulation process is shown in Figure 3.25 and the original NC file is shown in Figure 3.26.

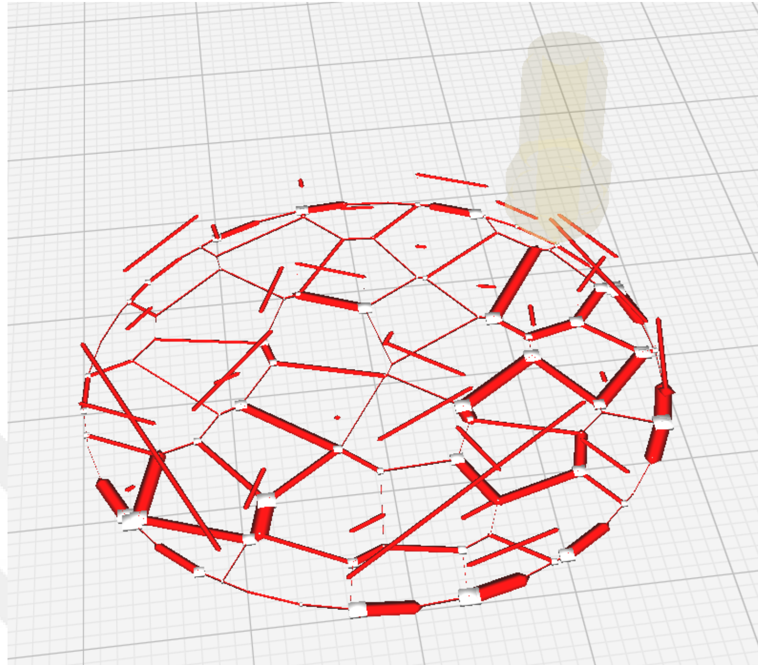


Figure 3.25: Simulation of the Original NC File in Cura Software

```
G92 E0
G1 X160.97 Y188.337 E0.10286 F600
G1 X160.97 Y188.337 Z0.25 F3000
G92 E0
G1 X149.119 Y183.762 E1.22219 F600
G1 X149.119 Y183.762 Z0.25 F3000
G92 E0
G1 X149.119 Y183.762 Z3.25 E0.28863 F600 ;target line
G1 X160.97 Y188.337 E1.51082
G1 X160.97 Y188.337 Z0.25 E1.79946
G1 X160.97 Y188.337 Z0.25 F3000
G92 E0
```

Figure 3.26: Segment of the Original NC File

The *NC Manipulation for Path Adjustment* Algorithm, Algorithm 13, is designed to manipulate the NC file from a given input file path ($inputPath$). The algorithm's primary objective is to adjust the path and ($retractionLength$) in the NC file. The NC lines are read and each line is iteratively processed. The core logic of the algorithm involves identifying specific G-code commands (specifically those starting with "G1"

and containing a "Z" value) and determining if there is a significant increase in the Z-axis value (more than 1mm). 1 mm is set as a threshold since the layer height is generally below 1mm and the user-specified height in Section 3.5.2 is more than 1 mm. When such a condition is met, the algorithm modifies the E value (extrusion length) of the previous line, applying the specified retraction length ($retractionLength$). Also, the identified target line and the next two lines are removed.

Furthermore, the algorithm addresses cases where there are unnecessary fast travels to the current location, resulting in lines with duplicated X and Y coordinates. These duplicated locations can cause extrusion to stop. In such cases, the algorithm omits adding the line to the modified NC file if it lacks extrusion (E value) and shares the same X and Y coordinates with the last processed line. After processing all lines, the algorithm writes the modified NC file to the specified output file path ($outputPath$).

The simulation in *Cura* for the first layer of manufacturing Cone geometry after the manipulation process is shown in Figure 3.27 and the original NC file is shown in Figure 3.28.

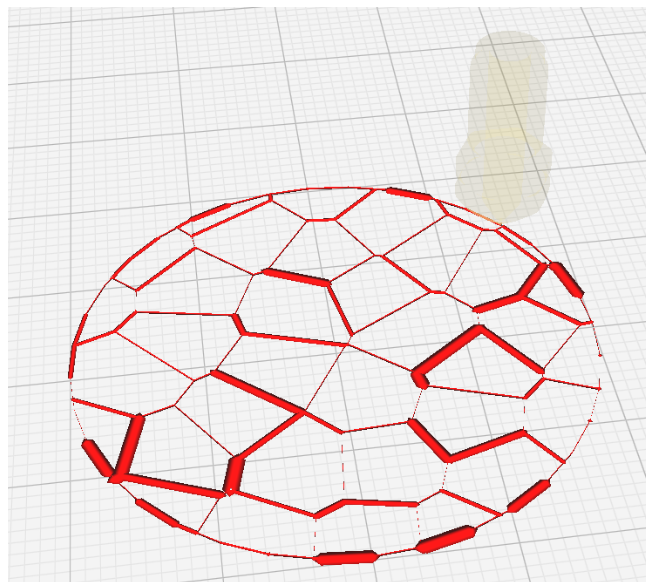


Figure 3.27: Simulation of the Manipulated NC File in Cura Software

```
G92 E0
G1 X106.772 Y103.876 E0.80959 F600
G92 E-1
G1 X106.9 Y103.888 Z0.24 F1800
G92 E0
G1 X110.767 Y110.451 E0.77363 F600
```

Figure 3.28: Segment of the Manipulated NC File

The NC File Manipulator Block is illustrated in Figure 3.29

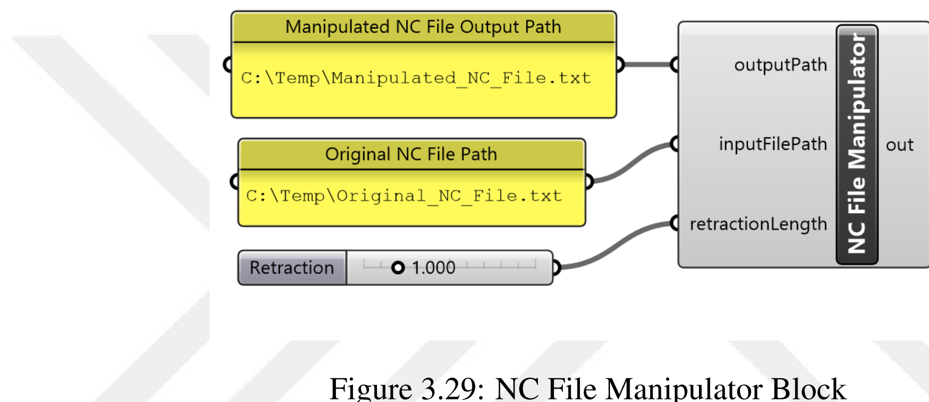


Figure 3.29: NC File Manipulator Block

3.5.5 Time Complexity Analysis

An empirical examination of time complexity was undertaken, focusing on the variable count of Voronoi seeds. Due to the challenging complexity of the analysis, analytical solutions were unfeasible. The number of Voronoi seeds was systematically varied from 0 to 7000 at intervals of 500, with corresponding execution times recorded. Subsequently, a graphical representation was generated and interpolation was performed using MS Excel. The data points are interpolated with a polynomial with second degree. The graph can be seen in Figure 3.30. The resultant algorithmic complexity was determined to be $\mathcal{O}(n^2)$.

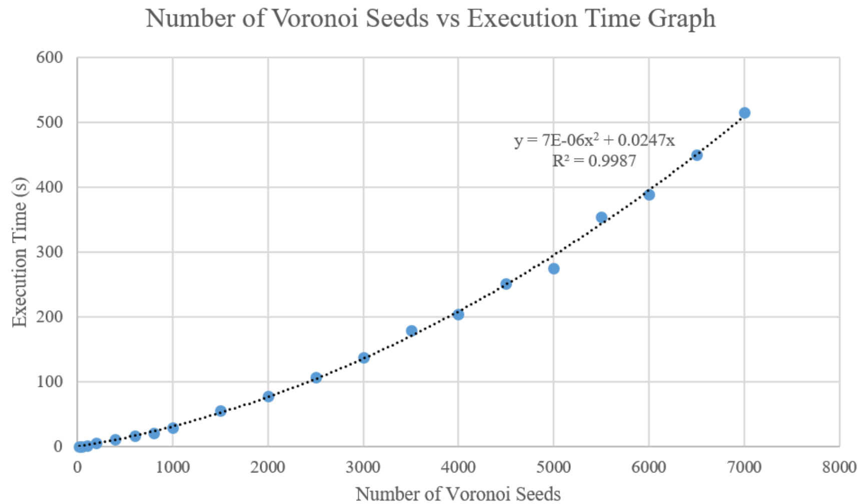


Figure 3.30: Number of Voronoi Seeds vs Execution Time Graph

3.5.6 2D Linear Interpolation

A methodology is employed to perform linear interpolation between the initial and the final slices of the provided Brep input, aimed at reducing execution time by processing only the initial and final slices of the BRep. This methodology is applicable under the condition that the number of vertices of the polylines at the initial and final slices are identical. Consequently, the polylines between the initial and final slices are lofted.

- The $offsetValue$ is multiplied by the length of the slices list to determine the weight values assigned to the cells at the final layer.
- The z-coordinates of the initial and final slices ($zslices$) are acquired by extracting the elements at indices 0 and -1 of the deconstructed domain.
- The boundary curves of the initial and final slices are retrieved by extracting the elements at indices 0 and -1 of the projected slices.
- The number of layers LN is designated as 2, reflecting the computation of only the initial and final layers.
- The Grasshopper components for pre-processing the input parameters are illustrated in Figure 3.31.

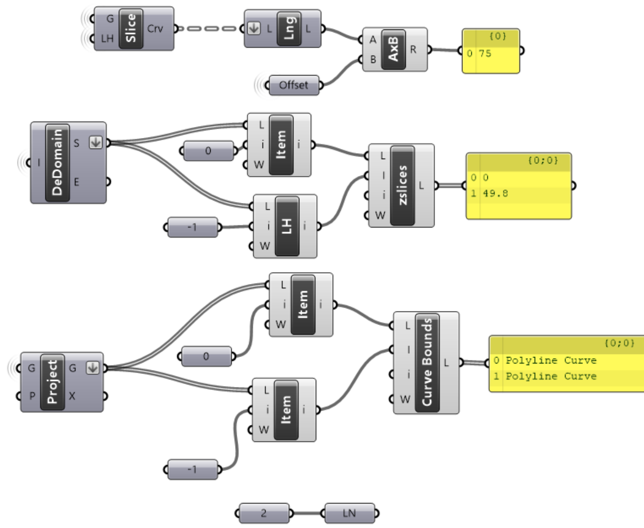


Figure 3.31: 2D Linear Interpolation Pre-Processing Blocks

The inputs are given to **Voronoi3DGenerator** Plug-In, resulting in the creation of a loft tree (LT), representing the customized arrangement of the resulting Voronoi polylines. Subsequently, the generated tree serves as input for the *Polylines Mesh Loft* block within the external *FroGH* Plug-In, initiating the 2D loft operation. Following the completion of the lofting process, a Grasshopper script is executed to verify the consistency between the number of items in the output of the *Polylines Mesh Loft* block and the user-defined number of Voronoi Seeds. The outcome of this verification process is displayed in a text panel.

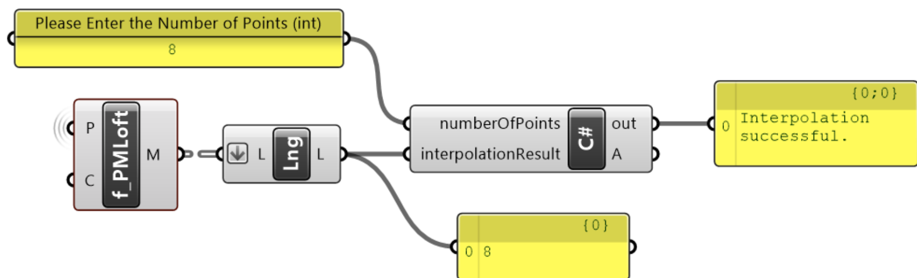


Figure 3.32: 2D Linear Interpolation and Verification Blocks

CHAPTER 4

IMPLEMENTATION & TEST CASES

In this chapter, the results of the proposed methodology are presented, including the visualizations and fabrications based on user-defined input parameters for different geometries, which are demonstrated respectively. Section 4.1 includes general methodology, while Section 4.2 encapsulates the outcomes for 2D Interpolation methodology.

4.1 Results and Visualizations for Voronoi3DGenerator

The comparison of the data sizes between the recently developed **.ont** format and the conventional STL format are illustrated in Table 4.1, emphasizing the proposed approach's data handling and representational efficiency. Due to the complex nature of the polyline Voronoi structures generated, they could not be transformed into surfaces; thus, to maintain consistency, the same number of Voronoi seeds was inserted across all target geometries using the built-in *Voronoi 3D* block. In the slicer software Cura, *Surface Mode* should be selected and Infill should be set to zero to print the exported STL with zero surface thickness.

The resultant geometries were exported in the binary and ASCII STL format, allowing for a direct comparison with the .ont file format in Kilobytes (kB). The proposed data format .ont undergoes compression via RAR archiving, chosen due to the compact binary formatting of the STL data, ensuring equitable comparison. The uncompressed .ont file is compared with the ASCII STL file. The comparison between data sizes of the proposed data format .ont and the conventional BRep data format STL for both compressed and uncompressed cases are illustrated in Table 4.1. The presented table

also details the input parameters such as layer height, offset, and seed count utilized by the **Voronoi3DGenerator** (see Algorithm 1) to provide visual representations and fabrication results. The result of the representation data size underscores the effectiveness of the proposed methodology. The parts have been fabricated with Creality Ender 3 V2 FDM machine. The manufacturing parameters are shown in Section 3.5.3, Figure 3.2.

Hyperlinks are included for each geometry, labeled as Cone (**1**), Truncated Pyramid (**2**), Large Truncated Pyramid (**3**), and Stanford Bunny (**4**) and Hypothetical Rectangular Prism (**5**).

Table 4.1: Comparative Results of File Sizes Based on Input Parameters for **Voronoi3DGenerator** Plug-In

	Target Geometries				
Specifications and Results	**1**	**2**	**3**	**4**	**5**
Layer Height (mm)	0.2	0.2	0.2	0.25	2
Offset (mm)	0.2	0.5	0.2	0.4	0.1
Number of Seeds	40	10	300	40	4000
.ont Data Size Uncompressed (kB)	26	4	13	68	140
.ont Data Size Compressed (kB)	2.8	0.7	6.1	15	71
ASCII STL Data Size (kB)	9893	649	17344	3820	392000
Binary STL Data Size (kB)	1440	98	2622	601	70200
Uncompressed Size Reduction (%)	99.74	99.38	99.93	98.22	99.96
Compressed Size Reduction (%)	99.81	99.34	99.77	97.50	99.90

• Cone

A truncated cone with 40 mm base, 30 mm top diameter and 30 mm height has been modeled and fabricated using the proposed methodology. The generated geometry and fabrication results are illustrated in Figures 4.1, 4.2, 4.4, and 4.5. The fabrication of the target geometry lasted 5 hours.

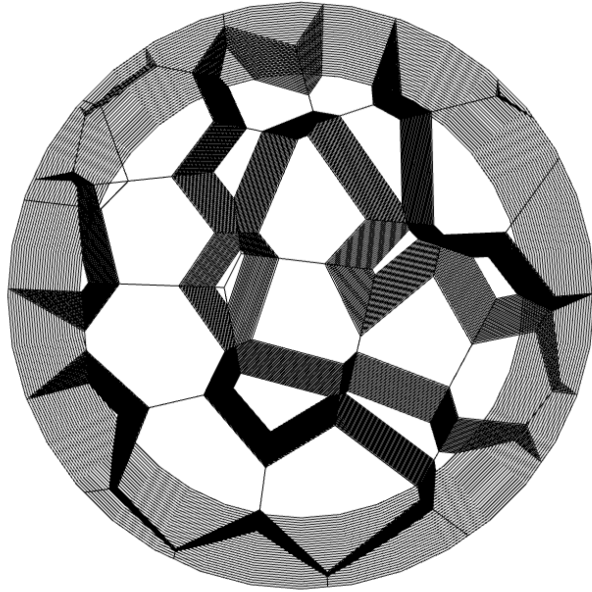


Figure 4.1: Top View of Generated Cone

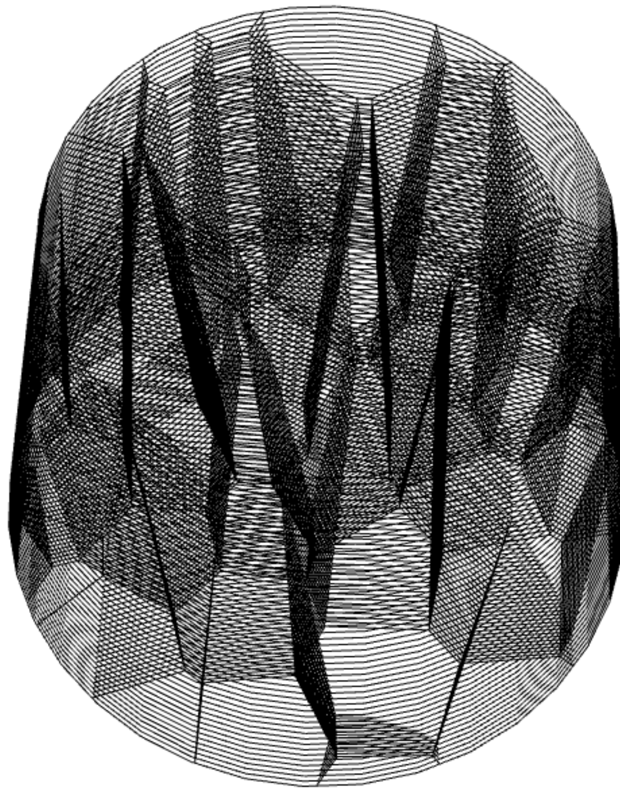


Figure 4.2: Perspective View of Generated Cone

Figure 4.3: Top View of Fabricated Cone

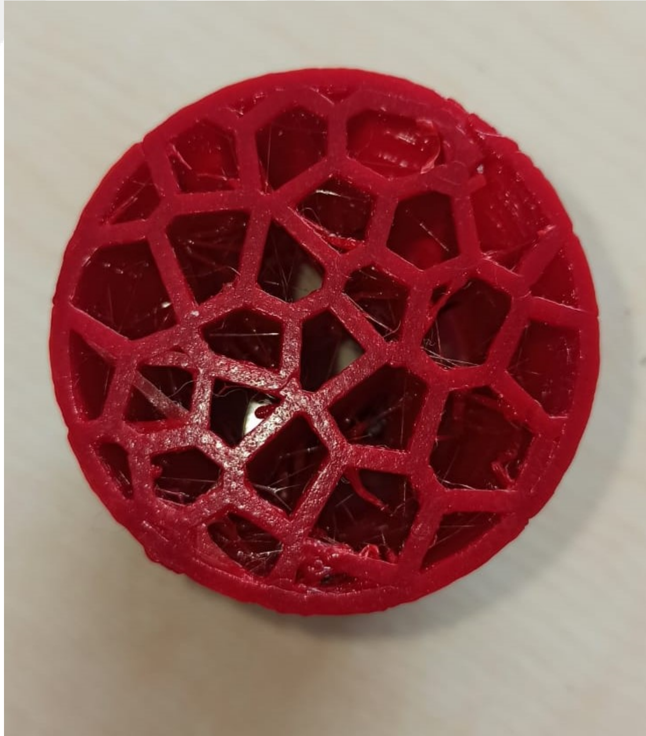


Figure 4.4: Bottom View of Fabricated Cone

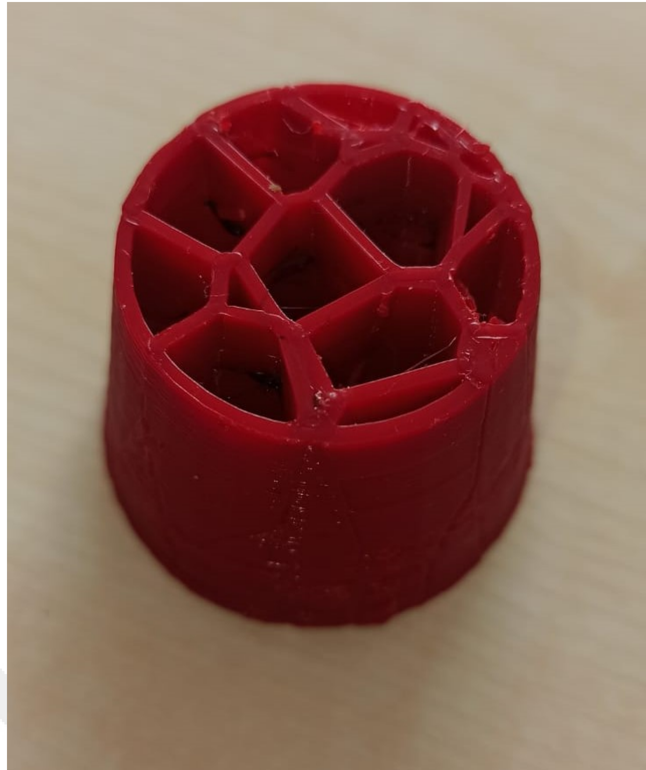


Figure 4.5: Perspective View of Fabricated Cone

- **Truncated Pyramid**

A truncated pyramid with $67mm \times 67mm \times 50mm$ dimensions 10°draft angle has been modeled and fabricated. The Voronoi polylines are meshed and NC file for the resulting STL geometry is generated in CURA software, unlike the other target geometries since the number of Voronoi polylines is relatively low and the resulting geometry is not complex. The results are illustrated in Figures 4.6, 4.7, 4.8 and 4.9. The fabrication of the target geometry took 6 hours.

- **Large Truncated Pyramid**

A truncated pyramid with dimensions $180mm \times 180mm \times 30mm$ and 10°draft angle has been modeled and fabricated to demonstrate the effectiveness of the proposed methodology in fabricating parts filled with a high number of Voronoi structures. The generated geometry and fabrication results are illustrated in Figures 4.10, 4.11 and 4.12 and 4.13. The fabrication of the target geometry took 29 hours.

Figure 4.6: Top View of Generated Truncated Pyramid

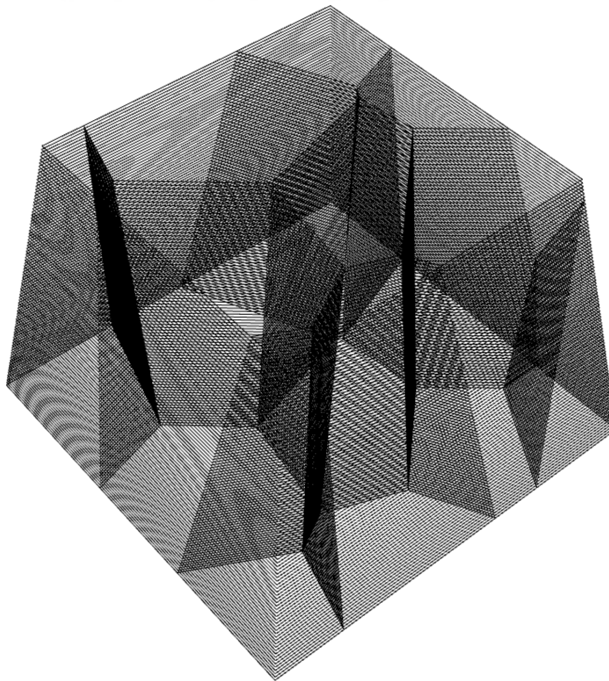


Figure 4.7: Perspective View of Generated Truncated Pyramid

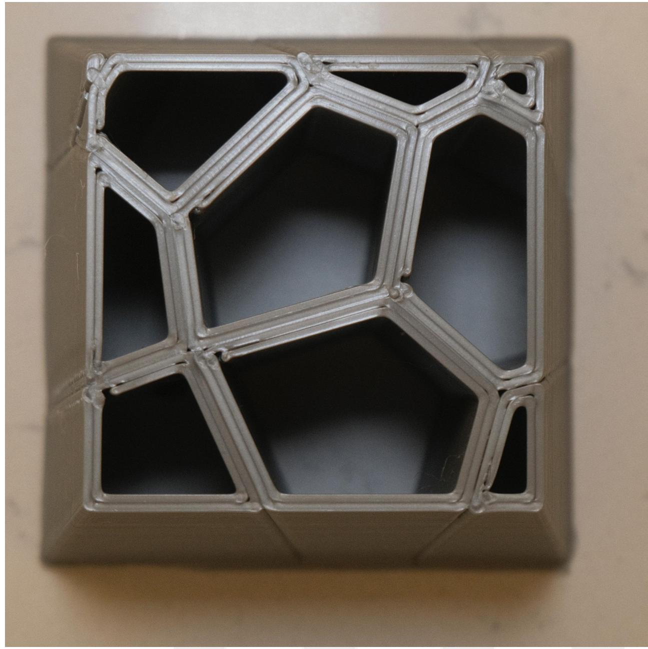


Figure 4.8: Top View of Fabricated Truncated Pyramid

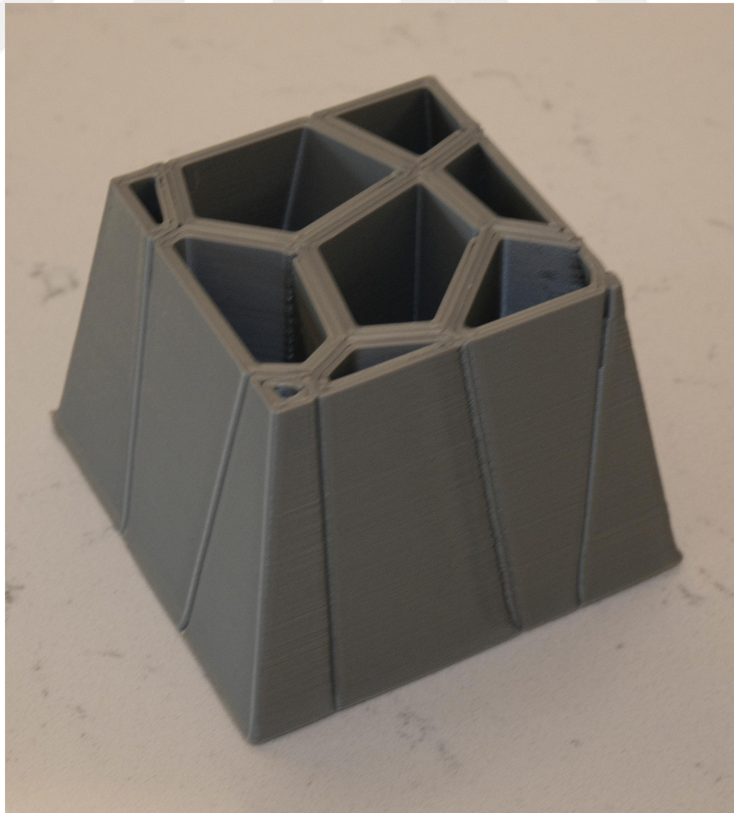


Figure 4.9: Perspective View of Fabricated Truncated Pyramid

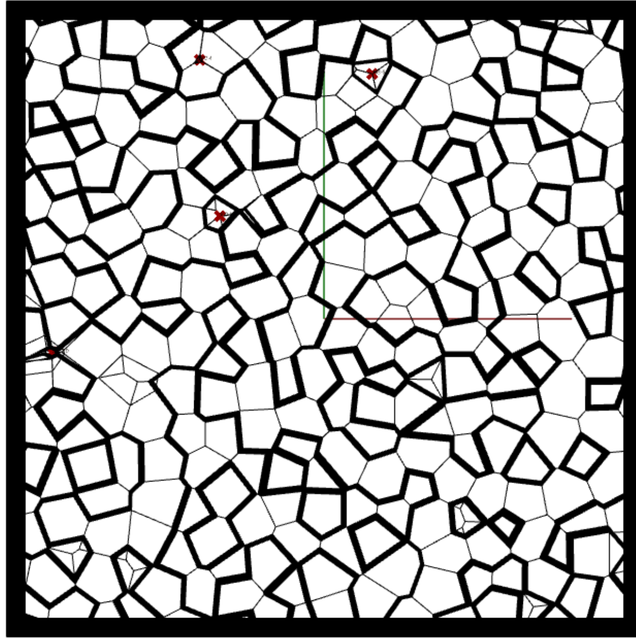


Figure 4.10: Top View of Generated Large Truncated Pyramid

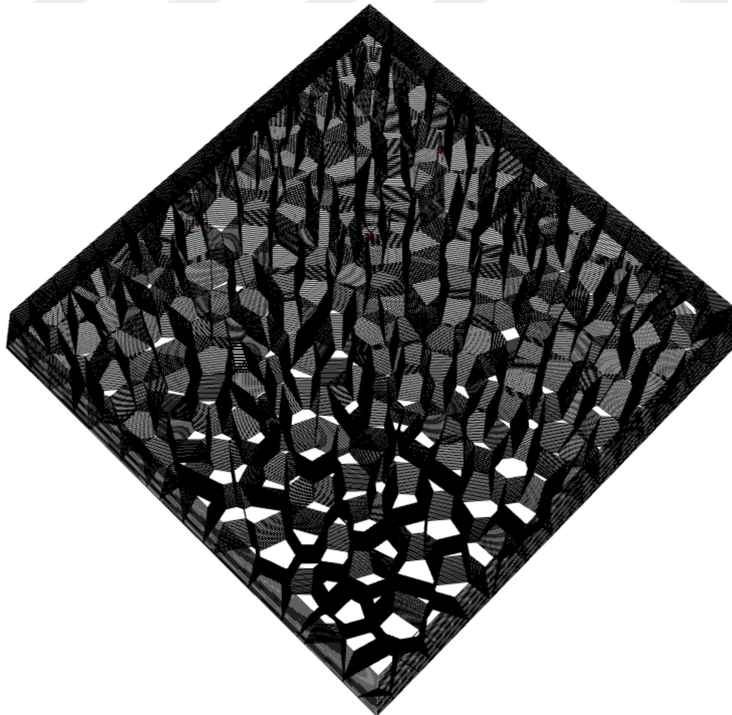


Figure 4.11: Perspective View of Generated Large Truncated Pyramid

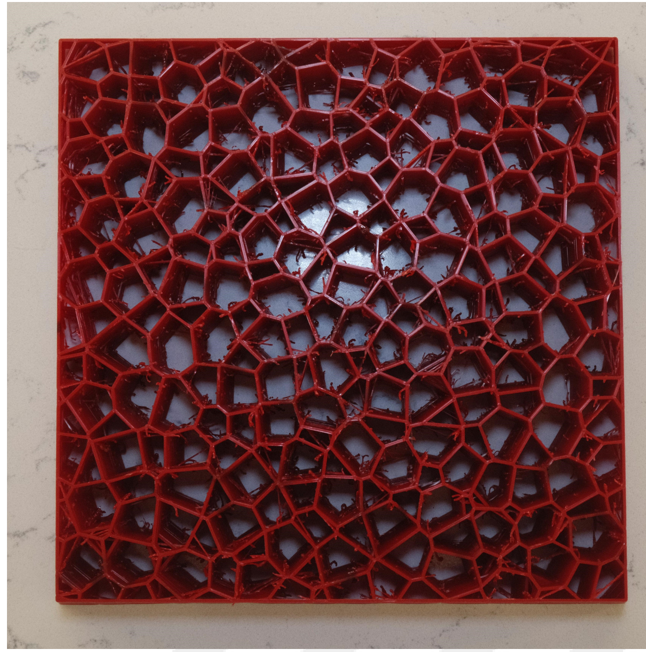


Figure 4.12: Top View of Fabricated Large Truncated Pyramid

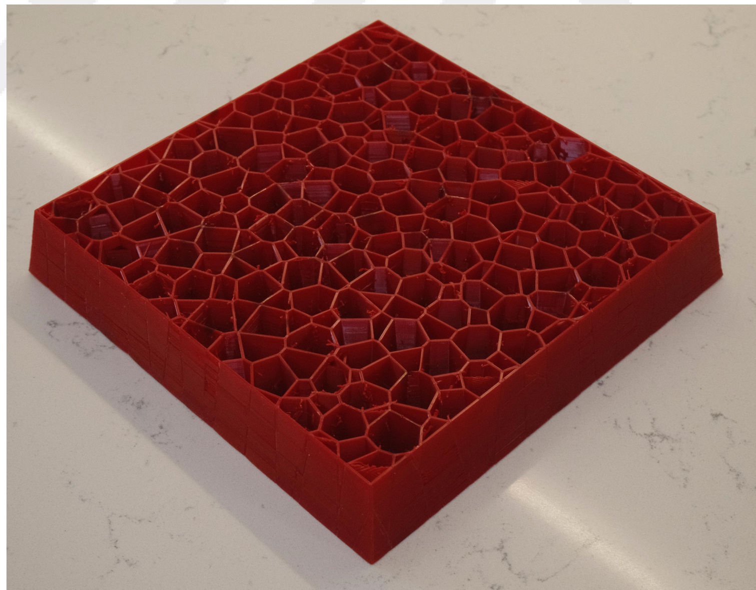


Figure 4.13: Perspective View of Fabricated Large Truncated Pyramid

- **Stanford Bunny**

Stanford Bunny is a challenging geometry since multiple cross-sectional areas exist in the same layer in some regions. It's proven that the proposed methodology is suitable

also for complex geometries. The generated Stanford Bunny geometry results are illustrated in Figures 4.14, 4.15, 4.16, and 4.17.

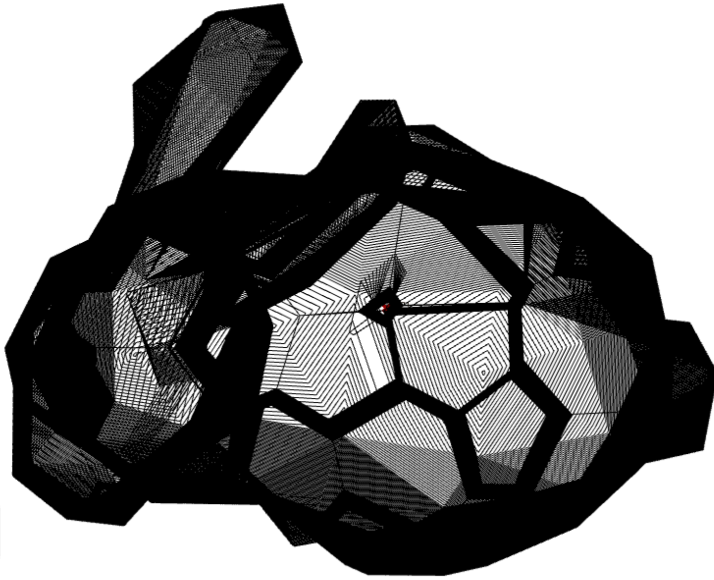


Figure 4.14: Top View of Generated Stanford Bunny

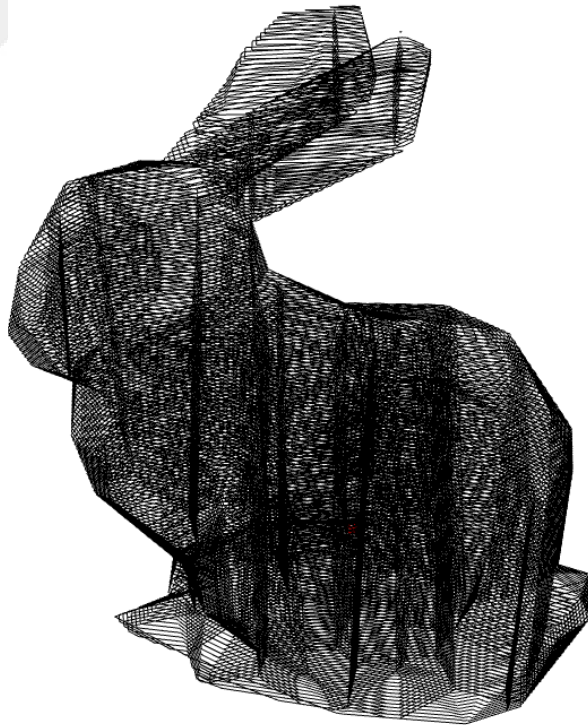


Figure 4.15: Perspective View of Generated Stanford Bunny

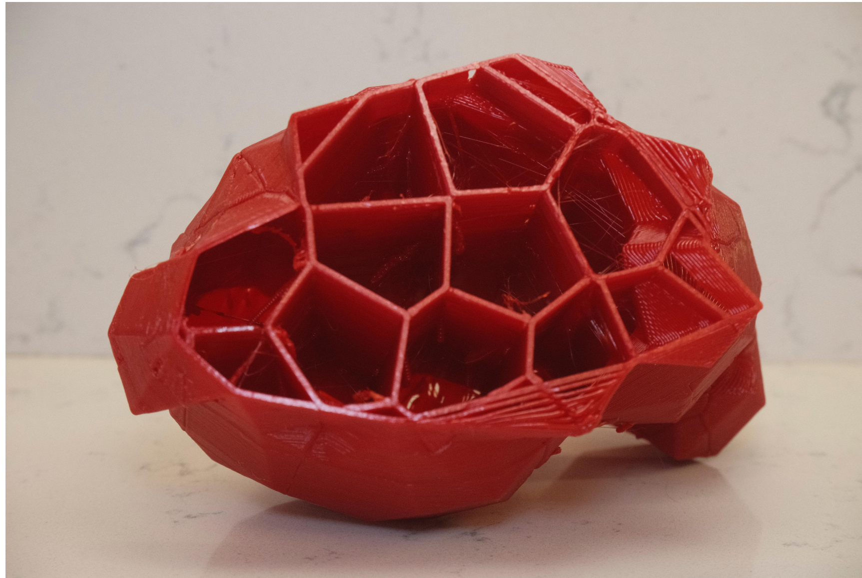


Figure 4.16: Bottom View of Fabricated Stanford Bunny

Figure 4.17: Perspective View of Fabricated Stanford Bunny

- **Hypothetical Rectangular Prism**

The Hypothetical Rectangular Prism geometry has been filled with 3D Voronoi based structures to prove the proposed methodology can be used to model the geometries in

large sizes filled with tiny internal structures as in the case of the DARPA TRADES competition [6]. The results of the generated rectangular prism with $1000mm \times 1000mm \times 40mm$ dimension are illustrated in Figures 4.18 and 4.19.

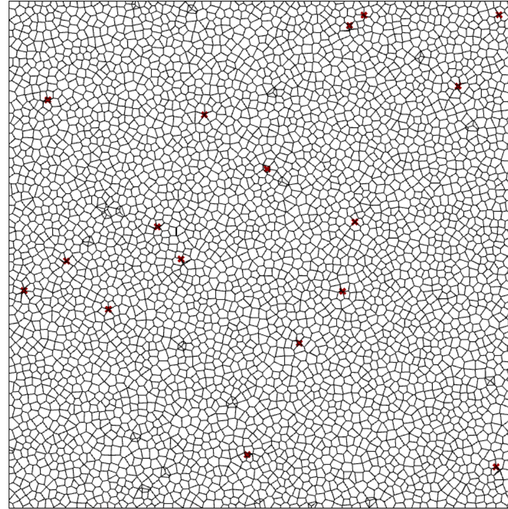


Figure 4.18: Top View of Generated Hypothetical Rectangular Prism

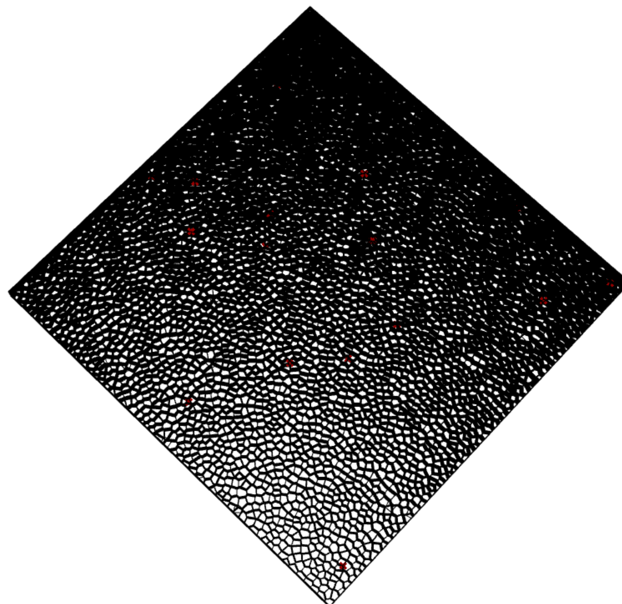


Figure 4.19: Perspective View of Generated Hypothetical Rectangular Prism

4.2 Results for 2D Linear Interpolation

The specifications and results for a cube with a 50 mm edge length using 2D Linear Interpolation methodology have been illustrated in Table 4.2.

Table 4.2: Comparative Results of Execution Times Based on Input Parameters for Voronoi3DGenerator Plug-In

	Target Geometry	
Specification and Results	**Cube**	**Cube**
Layer Height (mm)	0.2	0.1
Offset (mm)	0.3	0.15
Number of Seeds	8	8
Regular Execution Time (ms)	3100	6000
Execution Time with 2D Interpolation (ms)	18	16
Reduction Percentage (%)	99.42	99.73

The generated geometry results are illustrated in Figures 4.20 and 4.21.

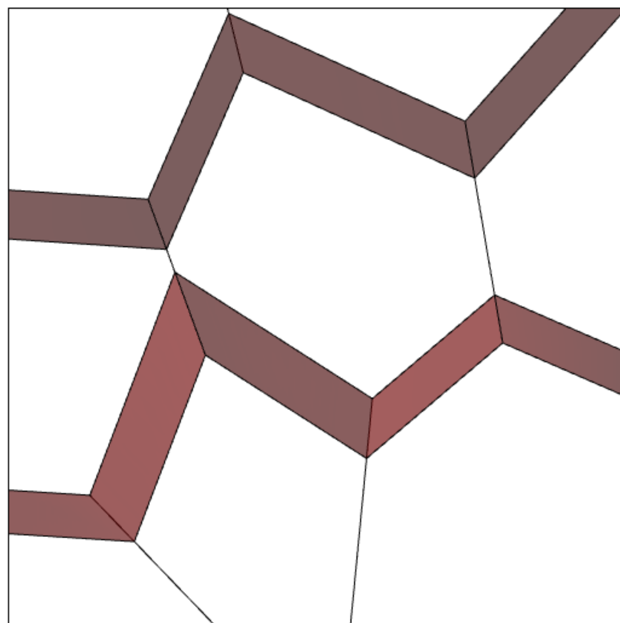


Figure 4.20: Top View of Interpolated Cube

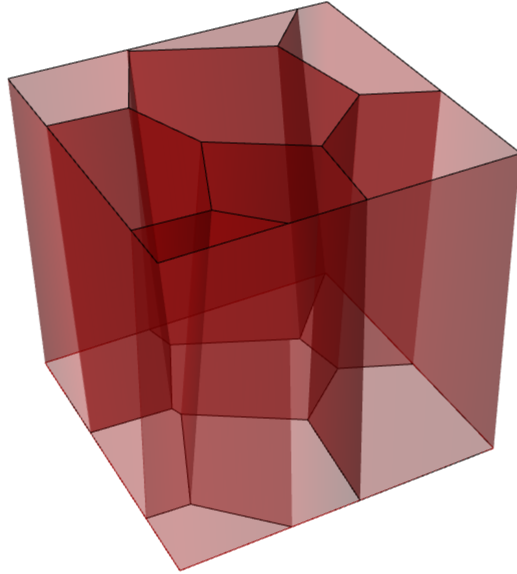


Figure 4.21: Perspective View of Interpolated Cube

CHAPTER 5

DISCUSSIONS & CONCLUSION

This thesis comprehensively investigates the efficient modeling, compact representation, and additive manufacturing of 3D Voronoi based internal structures derived from 2D Voronoi layers. The primary motivation underpinning this research was to develop a generative design algorithm capable of producing 3D Voronoi based internal structures through the utilization and manipulation of 2D Voronoi layers. A key aspect of this effort was the reduction of data size required to represent the generated structures while ensuring the exactness of the generated geometry. Additionally, a significant achievement of this research is the demonstration of successful geometry fabrication utilizing FFF machines. This process notably eliminates the requirement for support structures, effectively facilitating the production process and thereby representing a key conclusion of the proposed methodology.

Through the application of generative design and a compact representation of 3D Voronoi based internal structures methodology, the user inputs, *mesh*, *layer height*, *the number of Voronoi Seeds*, and *offset* are taken and the output geometry, the input mesh filled with 3D Voronoi based internal structures, is obtained. For the generation of the 3D Voronoi based internal structures, a *Rhinoceros Grasshopper* Plug-In, named **Voronoi3DGenerator**, is developed using *Visual Studio C# template for Grasshopper Assemblies*. The program aims to maintain constant Voronoi density through each layer of sliced geometry.

A new data format called **.ont** has been proposed for representing output geometry compactly. This proposed text-based format, generated by a custom C# script, includes *layer height*, *offset*, *Voronoi seed coordinates*, and the representation of the input mesh in the same format as an STL file. The size of the representation data is

significantly reduced compared to the conventional BRep data format, STL. When the size of the parts increases and the internal structures become smaller, the effectiveness and data reduction ratio are significantly improved. This is because the proposed methodology is primarily based on the initial layer data, and the subsequent layers are constructed generatively.

Moreover, the exactness of the output geometry is fully conserved since a resolution does not exist in the process of generating representation file. In other words, the exact geometry can be obtained as the .ont file is given as input to the **Voronoi3DGenerator** Plug-In. In addition to the script that generates **.ont** file, a custom script is also developed to interpret the information within the **.ont** file and provide them as input to the developed **Voronoi3DGenerator** Plug-In.

In addition to generative modeling and compact representation of parts filled with 3D Voronoi based internal structures, these parts are manufactured. Manufacturing is performed by using FFF machines. The required NC file for fabrication is generated in *Grasshopper* by using external *Droid* Plug-In. The NC File Generator Plug-In *Droid* does not generate NC files by considering the continuity of the paths; thus, extra materials and retractions are often observed during fabrication. To overcome these issues, the Voronoi cells are arranged considering the Minimum Spanning Tree of the corresponding layer.

Additionally, a continuous path generation algorithm is utilized. In this way, the continuous path where the material is continuously extruded from the nozzle is aimed to be maximized. This reduces the fabrication process's unwanted material and excessive number of retractions.

As *Droid* does not have built-in machine configurations, the specific commands for the target machines are not included in the generated NC file. Therefore, the Header and Footer of the NC file are obtained using an example NC file generated by a commonly used slicer software, *Ultimaker Cura*, as it has built-in machine configurations and can include machine-specific commands. The user takes an example NC file output from slicer software and adds markers inside the NC file as *START* and *END*. The lines before marker *START* and after marker *END* are collected and given as Header and Footer to *Droid*. This is because the preparation and finalizing commands are

generally machine-specific. The generated NC file is tested using slicer software to observe the simulated printing process. Finally, the NC file output of *Droid* was manipulated and the parts were fabricated using FFF machines.

As evident from the findings, the utilization of the **.ont** format leads to a substantial reduction in data size compared to the conventional Brep data format, STL. Notably, the effectiveness of the proposed methodology exhibits a significant enhancement with an escalation in the number of Voronoi seeds and the size of the respective part. This phenomenon is attributed to the direct dependence of the representation data in the **.ont**" format on the initial layer parameters and the boundary mesh of the target part. Consequently, an expansion in the size of the part and an increase in the number of Voronoi seeds do not proportionally escalate the data representation size. In contrast, conventional methods, such as STL, experience a considerable rise in data size with a significant increase in the number of meshes. The tabulated results in Table 4.1 illustrate a remarkable data size reduction up to 99.78%, underscoring the pronounced effectiveness of the presented methodology.

Moreover, a custom *2D Linear Interpolation* methodology is employed for simple geometries, where the number of vertices of the Voronoi cells at the initial and final layers remains constant. The primary objective is to compute only the initial and final layers of the slices, thereby reducing execution time. As demonstrated in Table 4.2, the execution time decreases by up to 99.73% for a simple cube geometry since instead of computing 500 layers, only 2 layers are computed. However, this interpolation methodology is infeasible for complex geometries where cross-section changes, and cases involve cell extinction or insertion.

In future developments of the **Voronoi3DGenerator** algorithm, an alternative approach to the post-offset operation for inserted points should be considered. The current methodology results in discontinuities during the post-offset process, leading to the necessity of preserving the original vertex count in the inserted cells. Consequently, these cells cannot initiate from a minimal size and expand continuously, a limitation not encountered in direct 3-D Voronoi diagram generation. Hence, there exists an opportunity for substantial enhancement of the algorithm in this aspect. The failure case where the preservation of the original vertex count is not considered is

illustrated in Figure 5.1. The figure on the left is the previous layer and the figure on the right is the current layer. The black lines show the cell after the post-offset operation, whereas the red lines show the cell before the post-offset operation. The blue circle shows where the discontinuity occurs between two layers when the number of vertices of the cell increases.

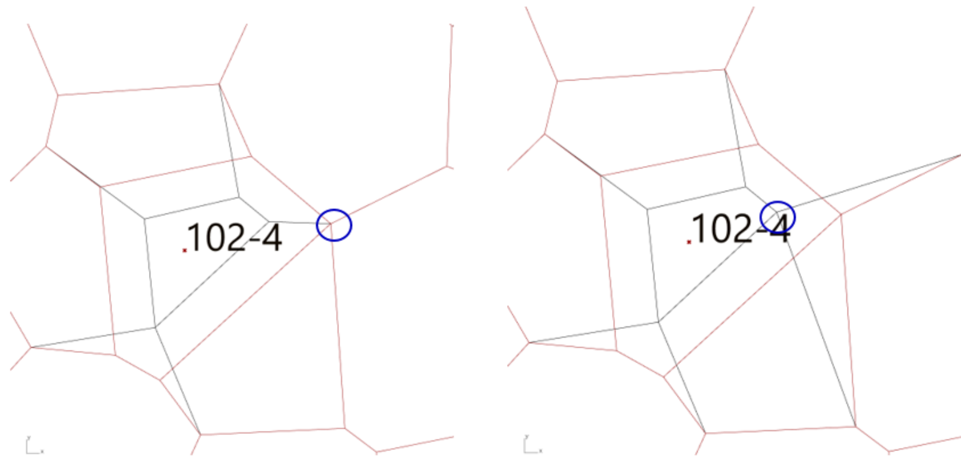


Figure 5.1: Failure Case where Discontinuity Occurs

Additionally, the fabrication of the resultant geometries presently does not produce a completely continuous path. While partial continuity is achieved, it is not complete, resulting in the presence of residual materials post-fabrication, which necessitates removal. The implementation of an algorithm similar to that discussed in the work of Özcan M. and Yaman U. [42] may offer a solution, enabling the generation of fully continuous paths for the internal 2D layers of the Voronoi structures. Also, the current methodology can produce parts with a single shell, which increases the potential risks of surface defects during the fabrication process.

Moreover, the utilization of implicit representation for the outer boundary could be explored as a further study. If the boundary mesh can be implicitly represented, it could lead to a further reduction in the data size of the .ont file, considering that the bulk of the data size is attributed to the BRep mesh within the file.

Furthermore, while the internal Voronoi structures do not require support structures, the boundary geometry might exhibit overhang areas. The current version of the

proposed algorithm cannot address such scenarios. Developing a specialized method for creating support structures in overhanging regions of the boundary geometry is thus a potential area for future enhancement.

In the current scope of this research, the generative design process is not configured to achieve specific target objectives, such as prescribed mass, maximal strength, or minimal compliance, primarily because the methodology lacks analysis tools and does not encompass iterative refinements. The geometry generation is directly dependent on the user-provided input parameters. Future work could enhance this process by incorporating iterative adjustments of parameters like *Number of Voronoi Seeds* and *offset*. Such advancements would enable the system to iteratively converge towards user-defined target objectives, thereby augmenting the functionality and adaptability of the generative design process.

REFERENCES

[1] J. Chen, G. Yang, and M. Yang, "Computation of compact distributions of discrete elements," *Algorithms*, vol. 12, 2019.

[2] P. Camps, M. Baes, and W. Saftly, "Using 3d voronoi grids in radiative transfer simulations," *Astronomy and Astrophysics*, vol. 560, 2013.

[3] D. Dobrovolskij and K. Schladitz, "Simulation of ultrasonic backscattering in polycrystalline microstructures," *Acoustics*, vol. 4, 2022.

[4] F. Bellelli, "The fascinating world of voronoi diagrams." `https://fbellelli.com/posts/2021-07-08-the-fascinating-world-of-voronoi-diagrams/`, 2021. Accessed: 2023-11-03.

[5] F. P. Preparata and M. I. Shamos, *Computational Geometry*. Springer-Verlag, 1985.

[6] "Computational prototyping group." `http://solidmodeling.org/trades-cp/`. Accessed: 2023-11-03.

[7] J. Boissonnat and M. Teillaud, "Effective computational geometry for curves and surfaces," 2006.

[8] G. Voronoi, "Nouvelles applications des paramètres conti-nus à la théorie de formes quadratiques," *Journal für die reine und angewandte Mathematik*, vol. 134, p. 198, 1908.

[9] B. Delaunay, "Title of the Article," *Classe des Sciences Mathématiques et Naturelles*, vol. 7, p. 793, 1934.

[10] F. Aurenhammer, "Power diagrams: Properties, algorithms and applications.," *SIAM Journal on Computing*, vol. 16, 1987.

[11] D. G. Abalde, "PowerDiagrams: A github repository," 2021.

[12] F. Laverne, F. Segonds, N. Anwer, and M. L. Coq, "Assembly based methods to support product innovation in design for additive manufacturing: an exploratory case study," *Journal of Mechanical Design*, vol. 137, 2015.

[13] P. Pradel and A. Rennie, "Future key research themes in design for additive manufacturing," 7 2021.

[14] S. Kim, D. W. Rosen, P. Witherell, and H. Ko, "A design for additive manufacturing ontology to support manufacturability analysis," *Journal of Computing and Information Science in Engineering*, vol. 19, 2019.

[15] M. L. McMillan, M. Jurg, M. Leary, and M. Brandt, "Programmatic generation of computationally efficient lattice structures for additive manufacture," *Rapid Prototyping Journal*, vol. 23, 2017.

[16] A. Wiberg, J. Persson, and J. Ölvander, "Design for additive manufacturing – a review of available design methods and software," *Rapid Prototyping Journal*, vol. 25, 2019.

[17] N. Chtioui, R. Gaha, and A. Benamara, "Design for additive manufacturing: Review and framework proposal," *Sustainable Engineering and Innovation*, vol. 5, 2023.

[18] O. Diegel, A. Nordin, and D. Motte, *DfAM Strategic Design Considerations*, pp. 41–70. 05 2019.

[19] X. Lorang, C. Mang, A. Tahmasebimoradi, and S. Girard, "Geometrical imperfections in lattice structures: a simulation strategy to predict strength variability," *World Congress in Computational Mechanics and ECCOMAS Congress*, vol. 1000, 2021.

[20] D. S. Nguyen, "Design of lattice structure for additive manufacturing in cad environment," *Journal of Advanced Mechanical Design Systems and Manufacturing*, 2019.

[21] L. Han, W. Du, Z. Xia, B. Gao, and M. Yang, "Generative design and integrated 3d printing manufacture of cross joints," *Materials*, vol. 15, 2022.

[22] H. Isakhani, N. Bellotto, Q. Fu, and S. Yue, "Generative design and fabrication of a locust-inspired gliding wing prototype for micro aerial robots," *Journal of Computational Design and Engineering*, vol. 8, 2021.

[23] M. K. Thompson, G. Moroni, T. H. Vaneker, G. Fadel, R. Campbell, I. Gibson, A. Bernard, J. Schulz, P. Graf, B. Ahuja, and F. Martina, "Design for additive manufacturing: trends, opportunities, considerations, and constraints," *CIRP Annals*, vol. 65, pp. 737–760, 2016.

[24] P. Murdy, J. Dolson, D. A. Miller, S. Hughes, and R. Beach, "Leveraging the advantages of additive manufacturing to produce advanced hybrid composite structures for marine energy systems," *Applied Sciences*, vol. 11, p. 1336, 2021.

[25] J. F. P. Lovo, I. L. d. Camargo, L. A. O. Araujo, and C. A. Fortulan, "Mechanical structural design based on additive manufacturing and internal reinforcement," *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, vol. 234, pp. 417–426, 2019.

[26] J. Raynaud, V. Pateloup, M. Bernard, D. Gourdonnaud, D. Passerieux, D. Cros, V. Madrangeas, P. Michaud, and T. Chartier, "Hybridization of additive manufacturing processes to build ceramic/metal parts: example of htcc," *Journal of the European Ceramic Society*, vol. 41, pp. 2023–2033, 2021.

[27] J. Um, J. Park, and I. A. Stroud, "Squashed-slice algorithm based on step-nc for multi-material and multi-directional additive processes," *Applied Sciences (Switzerland)*, vol. 11, 2021.

[28] M. K. Thompson, G. Moroni, T. H. Vaneker, G. Fadel, R. Campbell, I. Gibson, A. Bernard, J. Schulz, P. Graf, B. Ahuja, and F. Martina, "Design for additive manufacturing: trends, opportunities, considerations, and constraints," *CIRP Annals*, vol. 65, pp. 737–760, 2016.

[29] E. Pei, M. Ressin, R. I. Campbell, B. Eynard, and J. Xiao, "Investigating the impact of additive manufacturing data exchange standards for re-distributed manufacturing," *Progress in Additive Manufacturing*, vol. 4, 2019.

[30] U. Yaman, M. Dolen, U. M. Dilberoglu, and B. Gharehpapagh, "A new method

for generating image projections in dlp-type 3d printer systems," *Procedia Manufacturing*, vol. 11, 2017.

[31] S. Liu, T. Liu, Q. Zou, W. Wang, E. L. Doubrovski, and C. C. Wang, "Memory-efficient modeling and slicing of large-scale adaptive lattice structures," *Journal of Computing and Information Science in Engineering*, vol. 21, 2021.

[32] B. Vaissier, J. P. Pernot, L. Chougrani, and P. Véron, "Lightweight mesh file format using repetition pattern encoding for additive manufacturing," *CAD Computer Aided Design*, vol. 129, 2020.

[33] X. Zhao and J. Huang, "Lightweight geometric compression encoding for additive manufacturing," *CAD Computer Aided Design*, vol. 165, 12 2023.

[34] nTopology Inc., "Implicit interop." `https://www.ntop.com/software/capabilities/implicit-interop/`, 2023. Accessed: 2023-11-11.

[35] S. Gómez, M. D. Vlad, J. López, and E. Fernández, "Design and properties of 3d scaffolds for bone tissue engineering," *Acta Biomaterialia*, vol. 42, 2016.

[36] H. Ledoux and C. M. Gold, "Modelling three-dimensional geoscientific fields with the voronoi diagram and its dual," *International Journal of Geographical Information Science*, vol. 22, pp. 547–574, 5 2008.

[37] Z. Li, D. Feng, B. Li, D. Xie, and Y. Mei, "Fdm printed mxene/mnfe2o4/mwcnts reinforced tpu composites with 3d voronoi structure for sensor and electromagnetic shielding applications," *Composites Science and Technology*, vol. 231, 2023.

[38] F. Feng, S. Xiong, Z. Liu, Z. Xian, Y. Zhou, H. Kobayashi, A. Kawamoto, T. Nomura, and B. Zhu, "Cellular topology optimization on differentiable voronoi diagrams," *International Journal for Numerical Methods in Engineering*, vol. 124, pp. 282–304, 1 2023.

[39] E. Mele, M. Fraldi, G. M. Montuori, G. Perrella, and V. D. Vista, "Hexagrid-voronoi transition in structural patterns for tall buildings," *Frattura ed Integrita Strutturale*, vol. 13, 2019.

[40] S. McMains, J. Smith, J. Wang, and C. Sequin, “Layered manufacturing of thin-walled parts,” *Proceedings of the ASME Design Engineering Technical Conference*, vol. 2, 2000.

[41] “Join multiple closed polygons in one line.” `https://www.grasshopper3d.com/forum/topics/join-multiple-closed-polygons-in-one-line`. Accessed: 2023-11-03.

[42] M. Özcan and U. Yaman, “A continuous path planning approach on voronoi diagrams for robotics and manufacturing applications,” *Procedia Manufacturing*, vol. 38, 2019.

APPENDIX A

PSEUDOCODES OF ALGORITHMS

Pseudocodes of the algorithms used and explained in Chapter 3 are provided in this chapter. The symbolics of the variables defined in Table 3.1 are used to simplify the notation.

The symbolics of the variables defined in Table 3.1 are used to simplify the notation in the following pseudocodes.

Algorithm 1 3D Voronoi Generator Algorithm

Inputs: Point List PL, Number of layers ln, z coordinates of slices $zslices$, $offset$, Curve Bounds CB, layer height lh

Outputs: NT, IPL, POT, MST, DL, VT, CVT

1: **for** each point in PL **do**
2:     Add 0 to OL
3: **end for**
4: **for** each point in PL **do**
5:     Add 0 to WL
6: **end for**
7: **for** lc from 0 to ln **do**
8:     Create a transformation vector $xform$ for Z coordinates based on $zslices[lc]$
9:     Clear SL
10:     **if** $lc > 0$ **then**
11:         **for** i from 0 to $WL.Count$ **do**
12:             **if** $OL[i] = 1$ **then**
13:                 Increase $WL[i]$ by offset
14:             **end if**
15:             **if** $OL[i] = -1$ **then**
16:                 Decrease $WL[i]$ by offset
17:             **end if**
18:         **end for**
19:     **end if**
20:     **if** $DL[lc-1] > DL[0]$ **then**
21:         Calculate sequence Call **GenerateSequence**VT
22:         Insert point Call **InsertPoint**(VT, S, PL , CB)
23:     **end if**
24:     Compute Delaunay triangulation at WORLDXY Plane by using **SuperDelaunay**
25:     Transform $CB[lc]$ to the current Z value of the layer using $xform$
26:     Transform Delaunay triangulation to the current Z value of the layer using $xform$
27:     Compute Voronoi diagram and trim it inside $CB[lc]$ using **SuperDelaunay**
28:     Add Voronoi to VT with branch lc
29:     Call **OptimizeWeight**(VT, PL, WL, CB) for inserted points

30: **for** i from 0 to delaunay.Vertices.Count **do**
31: Compute neighbouring topology by using **SuperDelaunay**
32: Add neighbouring topology of current $[lc]$ to NT
33: **end for**
34: Construct Minimum Spanning Tree using **SuperDelaunay**
35: Add Minimum Spanning Tree to SL for the current layer $[lc]$
36: **if** $lc = 0$ **then**
37: **for** i from 0 to $SL.Count - 1$ with step 2 **do**
38: **if** $WL[SL[i].Value] = 0$ **then**
39: Set $WL[SL[i+1].Value]$ to $WL[SL[i+1].Value] +$ offset
40: Set $OL[SL[i+1].Value]$ to 1
41: **end if**
42: **end for**
43: **end if**
44: Add SL to MST to obtain spanning tree of whole geometry
45: Arrange VT considering MST
46: Construct CVT with arranged VT
47: Calculate area of the current layer
48: Initialize $polycounter$ to 0
49: **for** i from 0 to $VT.Branch(lc).Count$ **do**
50: Get the polyline $\mathcal{P} = VT.Branch(lc)[i]$
51: **if** $\mathcal{P}$ is not null **then**
52: Increment $polycounter$ by 1
53: **end if**
54: **if** $area \neq 0$ **then**
55: Calculate density: $density = \frac{area}{polycounter}$
56: Add $density$ to DL
57: **end if**
58: **end for**
59: Call **Post-Offset Algorithm for Inserted Cells** (VT, PL , CB, OL, WL, NT)
60: **end for**
61: **return** NT, IPL, POT, MST, DL, VT, CVT

Algorithm 2 Proximity Check Between Polyline Points and a Curve

Inputs: A polyline $\mathcal{P}$, a curve C, a tolerance value τ

Output: Boolean value indicating if any point of $\mathcal{P}$ is within τ distance from C

1: Initialize a boolean variable a as false
2: **for** each point P_i in $\mathcal{P}$ **do**
3:     Get P_i from $\mathcal{P}$
4:     Find the closest point C_{closest} on C to P_i
5:     Calculate the distance d from P_i to C_{closest}
6:     **if** $d < \tau$ **then**
7:         Set a to true
8:         **break**
9:     **end if**
10: **end for**
11: **return** a

Algorithm 3 Precision Adjustment of 3D Point Coordinates

Inputs: A 3D point P, Number of decimal places n

Output: A new 3D point P' with coordinates rounded to n decimal places

1: Extract the X, Y, and Z coordinates of P
2: Round the X, Y, and Z coordinates of P to n decimal places
3: Create a new 3D point P' with the rounded coordinates
4: **return** P'

Algorithm 4 Distinct Point Enumeration in a Geometric Polyline

Input: A polyline $\mathcal{P}$

Output: The number of unique points in $\mathcal{P}$

1: Initialize an empty HashSet S for unique points
2: **for** each point P_i in $\mathcal{P}$ **do**
3:     **if** P_i is not already in S **then**
4:         Add P_i to S
5:     **end if**
6: **end for**
7: **return** the count of points in S

Algorithm 5 Symmetric Index Sequence Generation from a Voronoi Structure (GenerateSequence)

Input: A DataTree of Voronoi Polyline cells VT

Output: A list S representing a symmetric sequence of indices

1: Initialize an empty list S for the sequence
2: Get the count N of cells in the first branch of V
3: **for** each cell index i from 0 to $\frac{N}{2}$ **do**
4:     Add i to S
5:     Add the symmetric index $N - 1 - i$ to S
6: **end for**
7: **if** N is odd **then**
8:     Add the middle cell index $\frac{N}{2}$ to S
9: **end if**
10: **return** S

Algorithm 6 Point Insertion Algorithm (InsertPoint)

Inputs: VT, S, PL , CB, DL, OL, WL

Outputs: Updated PL, OL, WL, IPL

1: **if** Layer count L greater than 1 and previous layer's density in DL greater than the first layer's **then**
2: **for** each index i in S **do**
3: **if** OL at index i is 1 **then**
4: Retrieve polyline $\mathcal{P}$ from previous Voronoi layer VT_{L-1}
5: **if** $\mathcal{P}$ is null **then**
6: Continue to next iteration
7: **end if**
8: Check if $\mathcal{P}$ is on boundary Call **Proximity Check Between Polyline Points and a Curve**($\mathcal{P}$, CB)
9: **if** $\mathcal{P}$ on boundary **then**
10: Continue to next iteration
11: **end if**
12: **for** each segment in $\mathcal{P}$ **do**
13: Retrieve and round segment's starting point
14: Add rounded point to $CP1$
15: **end for**
16: **for** each neighbour index j of current cell **do**
17: Retrieve first neighbour polyline $\mathcal{P}_n$ from VT_{L-1}
18: **if** $\mathcal{P}_n$ is on boundary $\vee$ an inserted cell $\vee$ getting larger **then**
19: Continue to next neighbour
20: **end if**
21: **for** each segment in $\mathcal{P}_n$ **do**
22: Retrieve and round segment's starting point
23: Add rounded point to $CP2$
24: **end for**

25: **for** each second neighbour index excluding current one **do**
26: Retrieve and process second neighbour polyline $\mathcal{P}_o$
27: **if** $\mathcal{P}_o$ is on boundary $\vee$ an inserted cell $\vee$ getting larger **then**
28: Continue to next neighbour
29: **end if**
30: **for** each segment in $\mathcal{P}_o$ **do**
31: Retrieve and round segment's starting point
32: Add rounded point to $CP3$
33: **end for**
34: **if** intersection of $CP1$, $CP2$ and $CP3$ lists is non-empty **then**
35: Store intersection point as I
36: Jump to EndOfLoops
37: **end if**
38: **end for**
39: Clear $CP1$, $CP2$ and $CP3$ for next iteration
40: **end for**
41: **end if**
42: **end for**
43: **end if**
44: **label** EndOfLoops:
45: Add I to PL for next layer Voronoi generation
46: Add w to WL as the value of $(lc - 1) * offset$ for corresponding I
47: Add integer 1 to OL for corresponding I
48: Set w_i as (lc-1)*offset
49: Set OL_i as 0
50: **return** PL, OL, WL, IPL

Algorithm 7 Optimization of Weights in Voronoi Structure Generation Algorithm (OptimizeWeight)

Inputs: VT, PL, WL, CB

Outputs: Updated VT, WL

1: **for** each added point P_i in VT from a certain layer onwards **do**
2: Calculate index for P_i
3: **if** Voronoi cell VC exists for P_i **then**
4: **if** first-time creation of VC **then**
5: **repeat**
6: Reduce weight WL_i of P_i
7: Recalculate Delaunay and Voronoi structures
8: Check if VC still exists for P_i
9: **if** VC does not exist **then**
10: Revert the last weight change in WL_i
11: Recalculate Delaunay and Voronoi structures
12: Break the loop
13: **end if**
14: **until** minimum weight for P_i to create VC is found
15: **end if**
16: Determine the layer where P_i is added
17: **if** current layer is above the addition layer and VC exists **then**
18: Compare unique vertex count of current and initial layers for VC
19: Call **Distinct Point Enumeration in a Geometric Polyline**(VC)
20: **while** vertex count of VC is greater than the initial addition layer and cell exists **do**
21: Decrease weight WL_i of P_i to make its cell smaller
22: Update the OL_i of P_i as -1 to provide contraction in subsequent layers
23: Recalculate Delaunay and Voronoi structures

24: Update unique vertex count of VC

25: Call **Distinct Point Enumeration in a Geometric Polyline**(VC)

26: **end while**

27: **end if**

28: **end if**

29: **end for**

30: **return** VT, WL

Algorithm 8 Post-Offset Algorithm for Inserted Cells

Inputs: VT, PL , CB, OL, WL, NT

Output: Modified Voronoi structure VT with reduced cell size

1: Initialize uniOut, uniPoly to 0
2: Initialize count to the number of cells in the first branch of VT
3: Initialize POL with zeros for each cell in the first branch of VT
4: Set offsetResolution to 0.1
5: **for** each layer lc from 0 to ln **do**
6:     **for** each target point $targetpoint$ from $VT.Branch(0).Count$ to $VT.Branch(lc).Count$ **do**
7:         **if** Voronoi cell $VT.Branch(lc)[targetpoint]$ exists **then**
8:             Add Voronoi cell to TPL
9:             Calculate new origin and existing origin based on Z value of target point
10:             Define XY and Z planes for offset operations
11:             Initialize or update POL for the target point
12:             **if** first-time operation for target point **then**
13:                 Initialize offset for the target point in POL
14:                 **repeat**
15:                     Perform offset operation on target Voronoi cell using *Clipper* Plug-In
16:                     Calculate number of vertices after offset
17:                     Calculate area of the resulting cell after offset
18:                     **if** resulting cell does not exist **OR** number of vertices decreased **OR** area < 2 mm^2 **then**
19:                         Perform offset in negative direction
20:                         Finalize cell shape and break loop
21:                     **end if**
22:                     Increase offset by resolution
23:                 **until** The Loop is break

```
24:         else
25:             Perform offset operation for previously handled points
26:         end if
27:         Collect vertices of the cells adjacent to the target cell
28:         Check whether the vertices are on the boundary (CB)
29:         if neighbour cell's vertex is not on the boundary then
30:             Transform the vertex to the closest point of the offset cell
31:         end if
32:     else
33:         Do not transform the vertex
34:     end if
35:     Process each vertex of target polyline for boundary check
36:     if target polyline's vertex is on the boundary then
37:         Add this vertex to the convexhulllist
38:     else
39:         Add the closest vertex of the offset cell to the convexhulllist
40:     end if
41:     Compute convex hull with the points convexhulllist in and update offset
    point list
42:     Clear auxiliary lists for the next iteration
43:   end for
44: end for
45: return VT
```

Algorithm 9 Exporting .ont Data Format

Inputs: Offset value ($offsetValue$), layer height (lh), seed coordinates list (PL), mesh structure ($mesh$), output file path ($outputPath$)

Outputs: Output content written to file, success message stored in A

1: **if** $mesh$ is null or invalid **then**
2: Set A to "Invalid mesh provided."
3: **return**
4: **end if**
5: Initialize content string ($content$) to empty
6: Append "[OFFSET_VALUE]\n" and $offsetValue$ to $content$
7: Append "[LAYER_HEIGHT]\n" and lh to $content$
8: Append "[VORONOI_SEED_COORDINATES]\n" to $content$
9: **for** each point P in PL **do**
10: Append coordinates of P to $content$
11: **end for**
12: Append "[BREP_PART_STL]\n" and "solid mesh\n" to $content$
13: Convert quad faces of $mesh$ to triangles
14: Compute face and vertex normals of $mesh$
15: Compact $mesh$
16: **for** each face i in $mesh$ **do**
17: Define $face$ as $mesh$ faces at index i
18: **if** indices of $face$ are valid **then**
19: Append facet normal and vertex coordinates to $content$
20: **end if**
21: **end for**
22: Append "endsolid mesh\n" to $content$
23: **if** $outputPath$ is not empty **then**
24: Write $content$ to file at $outputPath$
25: Set A to "File exported successfully!"
26: **else**
27: Set A to "Please provide a valid output path."
28: **end if**

Algorithm 10 Importing .ont Data Format

Input: File path ($filePath$)

Outputs: Seed coordinates list (PL), Offset value ($offsetValue$), Layer height (lh), Reconstructed mesh ($meshOut$)

1: Initialize PL as an empty list of Point3D
2: Initialize $offsetValue$ and lh to 0
3: Initialize $meshOut$ as an empty Mesh
4: **if** $filePath$ is empty or file does not exist **then**
5: Print "Invalid file path provided."
6: Set outputs PL, $offsetValue$, lh to null/0 and return
7: **end if**
8: Read all lines from the file at $filePath$ into $lines$
9: **for** each line i in $lines$ **do**
10: Trim the line and store in $line$
11: **if** $line$ equals "[OFFSET_VALUE]" **then**
12: Parse next line as $offsetValue$ and increment i
13: **else if** $line$ equals "[LAYER_HEIGHT]" **then**
14: Parse next line as lh and increment i
15: **else if** $line$ equals "[VORONOI_SEED_COORDINATES]" **then**
16: Increment i and parse subsequent lines for seed coordinates until next section
17: **else if** $line$ starts with "solid" **then**
18: Increment i and parse subsequent lines for mesh data until "endsolid"
19: **end if**
20: **end for**
21: Assign PL, $offsetValue$, lh, and $meshOut$ to corresponding outputs

Algorithm 11 Eliminate Duplicate Lines in Voronoi Structure

Input: Data tree of Polylines VT

Output: Data tree of unique Curves $resultTree$

```
1: Initialize a new data tree resultTree for Curves
2: for each branch path in VT do
3:     Initialize a list uniqueCurves for storing unique Curves
4:     Get the list of Polylines polylines from VT at path
5:     for each polyline in polylines do
6:         if polyline is not null then
7:             Convert polyline to a NURBS Curve nurbsCurve
8:             if nurbsCurve is not null then
9:                 Duplicate nurbsCurve into constituent Curves curves
10:                for each curve in curves do
11:                    if curve is not already in uniqueCurves then
12:                        Add curve to uniqueCurves
13:                    end if
14:                end for
15:            end if
16:        end if
17:    end for
18:    for each uniqueCurve in uniqueCurves do
19:        Add uniqueCurve to resultTree at path
20:    end for
21: end for
22: return resultTree
```

Algorithm 12 Example NC File Segregation Based on Line Markers

Input: File path ($filePath$)

Outputs: Prepended text ($prependedText$), Appended text ($appendedText$)

1: Read all lines from the file at $filePath$ into $lines$
2: Initialize $prependList$ and $appendList$ as empty lists of strings
3: Initialize $lastG1Index$ and $firstG1Index$
4: Find the first index of a line starting with "START" and store in $firstG1Index$
5: Find the last index of a line starting with "END" and store in $lastG1Index$
6: **for** each line i in $lines$ **do**
7:     **if** i is less than $firstG1Index$ **then**
8:         Add line i to $prependList$
9:     **else if** i is greater than $lastG1Index$ **then**
10:         Add line i to $appendList$
11:     **end if**
12: **end for**
13: Set $prependedText$ to the contents of $prependList$
14: Set $appendedText$ to the contents of $appendList$
    return ($prependedText$), ($appendedText$)

Algorithm 13 NC File Manipulation for Path Adjustment

Input: Output file path ($outputPath$), Input file path ($inputPath$), Length of retraction ($retractionLength$)

Output: Modified NC File saved to $outputPath$

1: **if** $inputPath$ does not exist **then**
2:     Exit
3: **end if**
4: Read all lines from $inputPath$ into gcodeLines
5: Initialize an empty list modifiedLines
6: Initialize previousZ as NaN
7: Initialize linesToSkip as 0
8: Initialize lastX and lastY as NaN
9: **for** each line in gcodeLines **do**
10:     **if** linesToSkip is greater than 0 **then**
11:         Decrement linesToSkip
12:         Continue to next iteration
13:     **end if**
14:     **if** line starts with "G1" and contains "Z" **then**
15:         Extract Z value from line
16:         **if** Z value increased more than 1mm from previousZ **then**
17:             **if** modifiedLines is not empty **then**
18:                 Modify E value of last line in modifiedLines using $retractionLength$
19:             **end if**
20:             Set linesToSkip to 2
21:             Update previousZ with current Z value
22:             Continue to next iteration
23:         **end if**
24:         Update previousZ with current Z value
25:     **end if**

26: Extract X and Y values from line

27: **if** line starts with "G1" and does not contain "E" and (lastX, lastY) is the same as (currentX, currentY) **then**

28: Continue to next iteration (Skip lines with same X and Y coordinates and no extrusion)

29: **end if**

30: Add line to modifiedLines

31: Update lastX and lastY with current X and Y values

32: **end for**

33: Write modifiedLines to $outputPath$