2016

Shape Design and Optimization for 3D Printing

Yahan Zhou

University of Massachusetts, Amherst

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_2

Part of the Graphics and Human Computer Interfaces Commons

Recommended Citation
https://scholarworks.umass.edu/dissertations_2/613

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.
SHAPE DESIGN AND OPTIMIZATION FOR 3D PRINTING

A Dissertation Presented

by

YAHAN ZHOU

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2016

College of Information and Computer Sciences
SHAPE DESIGN AND OPTIMIZATION FOR 3D PRINTING

A Dissertation Presented

by

YAHAN ZHOU

Approved as to style and content by:

________________________________________
Rui Wang, Chair

________________________________________
Evangelos Kalogerakis, Member

________________________________________
Erik Learned-Miller, Member

________________________________________
Ian R. Grosse, Member

________________________________________
James Allan, Chair
College of Information and Computer Sciences
ACKNOWLEDGMENTS

First of all, I must express my gratitude to my advisor, Prof. Rui Wang, for his help and advice during my Ph.D. study. We have worked together in many different projects, and he has been offering me great guidance in all those projects. Then I thank my other defense committee members, including Prof. Evangelos Kalogerakis, Prof. Erik Learned-Miller and Prof. Ian R. Grosse. They supervised the whole process of my thesis writing. I also thank people who I have collaborated with, including Haibin Huang, Li-Yi Wei, Ling-Qi Yan, Kun Xu, Zhaoliang Lun, Jacqueline Feild, Shinjiro Sueda, Wojciech Matusik, Ariel Shamir, Min Jiang, Richard Southern, Jian Jun Zhang. Thanks to Leeanne Leclerc for her generous help to me and all computer science students. And thanks to Prof. Chen Jin, who recommended me to join UMass.

I also thank my parents. Without their support it would be impossible for me to finish my Ph.D degree. Thanks all my friends and fellow students at UMass, Amherst. I had a great time with them. At last, special thanks to my wife, Ying Cao, without whom I would have a different graduation time.
ABSTRACT

SHAPE DESIGN AND OPTIMIZATION FOR 3D PRINTING

FEBRUARY 2016

YAHAN ZHOU
B.Sc., FUDAN UNIVERSITY, SHANGHAI, CHINA
M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST
Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Rui Wang

In recent years, the 3D printing technology has become increasingly popular, with widespread uses in rapid prototyping, design, art, education, medical applications, food and fashion industries. It enables distributed manufacturing, allowing users to easily produce customized 3D objects in office or at home. The investment in 3D printing technology continues to drive down the cost of 3D printers, making them more affordable to consumers.

As 3D printing becomes more available, it also demands better computer algorithms to assist users in quickly and easily generating 3D content for printing. Creating 3D content often requires considerably more efforts and skills than creating 2D content. In this work, I will study several aspects of 3D shape design and optimization for 3D printing. I start by discussing my work in geometric puzzle design, which is a popular application of 3D printing in recreational math and art. Given user-provided input figures, the goal is to compute the minimum (or best) set of geometric shapes that can satisfy the given constraints (such as dissection constraints). The puzzle design also has to consider feasibility, such as avoiding
interlocking pieces. I present two optimization-based algorithms to automatically generate customized 3D geometric puzzles, which can be directly printed for users to enjoy. They are also great tools for geometry education.

Next, I discuss shape optimization for printing functional tools and parts. Although current 3D modeling software allows a novice user to easily design 3D shapes, the resulting shapes are not guaranteed to meet required physical strength. For example, a poorly designed stool may easily collapse when a person sits on the stool; a poorly designed wrench may easily break under force. I study new algorithms to help users strengthen functional shapes in order to meet specific physical properties. The algorithm uses an optimization-based framework — it performs geometric shape deformation and structural optimization iteratively to minimize mechanical stresses in the presence of forces assuming typical use scenarios. Physically-based simulation is performed at run-time to evaluate the functional properties of the shape (e.g., mechanical stresses based on finite element methods), and the optimizer makes use of this information to improve the shape. Experimental results show that my algorithm can successfully optimize various 3D shapes, such as chairs, tables, utility tools, to withstand higher forces, while preserving the original shape as much as possible. To improve the efficiency of physics simulation for general shapes, I also introduce a novel, SPH-based sampling algorithm, which can provide better tetrahedralization for use in the physics simulator. My new modeling algorithm can greatly reduce the design time, allowing users to quickly generate functional shapes that meet required physical standards.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>x</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Design of Geometric Dissection Puzzles</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Design of Box-shape Geometric Puzzles</td>
<td>7</td>
</tr>
<tr>
<td>1.4 Optimization for Functional Shapes Design</td>
<td>11</td>
</tr>
<tr>
<td>1.5 Contributions</td>
<td>17</td>
</tr>
<tr>
<td>2. COMPUTATIONAL ALGORITHM FOR CREATING GEOMETRIC DISSECTION PUZZLES</td>
<td>18</td>
</tr>
<tr>
<td>2.1 Background and Related Work</td>
<td>20</td>
</tr>
<tr>
<td>2.2 Algorithms and Implementation</td>
<td>24</td>
</tr>
<tr>
<td>2.2.1 Assumptions and Overview</td>
<td>24</td>
</tr>
<tr>
<td>2.2.2 Dissecting Two Figures on a Square Lattice</td>
<td>25</td>
</tr>
<tr>
<td>2.2.3 Level 1 Optimization</td>
<td>27</td>
</tr>
<tr>
<td>2.2.4 Level $\ell$ Optimization</td>
<td>29</td>
</tr>
<tr>
<td>2.2.5 Area-Based Term</td>
<td>31</td>
</tr>
<tr>
<td>2.3 Extensions</td>
<td>34</td>
</tr>
<tr>
<td>2.3.1 Extension to the Triangular Lattice</td>
<td>34</td>
</tr>
<tr>
<td>2.3.2 Extension to 3D Shape Dissection</td>
<td>34</td>
</tr>
<tr>
<td>2.3.3 Dissecting Multiple Figures</td>
<td>35</td>
</tr>
<tr>
<td>2.4 Implementation Details</td>
<td>36</td>
</tr>
<tr>
<td>2.5 Results and Discussions</td>
<td>38</td>
</tr>
</tbody>
</table>
3. BOXELIZATION: FOLDING 3D OBJECTS INTO BOXES ....................... 42
   3.1 Related Work .............................................................. 43
   3.2 Methodology .............................................................. 45
      3.2.1 Voxelization ......................................................... 46
      3.2.2 Tree Fitting ......................................................... 48
      3.2.3 Interactive Folding ............................................... 55
   3.3 Results ................................................................. 56

4. DIRECT SHAPE OPTIMIZATION FOR STRENGTHENING PRINTABLE OBJECTS .......................................................... 61
   4.1 Background .............................................................. 63
      4.1.1 Physics Simulation .................................................... 63
      4.1.2 Surface Laplacian Distance ....................................... 68
   4.2 Related Work ............................................................ 72
   4.3 Overview ............................................................... 75
   4.4 Shape Optimization ..................................................... 78
      4.4.1 Elasticity Model ..................................................... 78
      4.4.2 Stress Function ...................................................... 80
      4.4.3 Geometric Constraints ............................................. 81
      4.4.4 Objective function ................................................ 83
      4.4.5 Optimization approach .......................................... 87
   4.5 Results ................................................................. 89
   4.6 Blue Noise Sampling using an SPH–based Method ..................... 95
      4.6.1 The Core Idea ....................................................... 96
      4.6.2 SPH Sampling and Implementation ................................ 101
      4.6.3 Varying Kernel Functions ....................................... 109
      4.6.4 Applications ....................................................... 112

5. CONCLUSIONS AND FUTURE WORK ......................................... 116

APPENDICES

A. PUBLICATION LIST .......................................................... 122
B. ANALYTIC GRADIENT AND HESSIAN OF STRAIN TENSOR \( \epsilon \) ........... 124
C. ANALYTIC HESSIAN OF ELASTIC ENERGY \( U \) ................................ 126
D. ANALYTIC GRADIENT AND HESSIAN OF VON MISES STRESS .......... 127
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Two dissection results from my algorithm. In each example, a set of puzzle pieces can be rearranged into construct both the left and right figures. The pieces are color-coded to indicate the solutions. Left: demonstration of the Pythagorean triple numbers $3^2 + 4^2 = 5^2$. Right: a heart figure that can be rearranged into a key figure. My algorithm computes each example with as few pieces as it can (i.e. minimum solution). The resulting solution can be used in applications such as recreational art and geometry education.</td>
</tr>
<tr>
<td>1.2</td>
<td>Dissection puzzles created using my algorithm and 3D printed into physical objects. The top two rows show 2D examples, each can construct three different shapes. The bottom two rows are 3D examples, with the pieces shown in the first column and two shapes constructed from them shown in the next two columns.</td>
</tr>
<tr>
<td>1.3</td>
<td>A classic example of dissection using only two pieces.</td>
</tr>
<tr>
<td>1.4</td>
<td>Folding a car into a cube. Our system finds a collision-free folding sequence.</td>
</tr>
<tr>
<td>1.5</td>
<td>An illustration of my method. (a) We first find the best voxelization of the input shape. (b) Geometric neighbors define the connectivity graph with nodes as voxels and edges as potential hinge locations. (c) We turn the graph into a tree. Some edges are removed, and some edges are turned into rigid links. The rest are assigned a joint type and a folding angle. (d) Once we compute the locations of the joints and their angles, the shape can transform into a box.</td>
</tr>
<tr>
<td>1.6</td>
<td>Illustration of a folding template.</td>
</tr>
<tr>
<td>1.7</td>
<td>Comparison of surface reconstruction using Bilateral Sampling [31] vs our method. The surface of the bowl is remeshed with 5000 samples. Note how our method preserves sharp features.</td>
</tr>
</tbody>
</table>
2.1 Four example sets of dissection puzzles created using our algorithm. The top two rows show 2D dissections – for each set we show three constructed figures. The bottom two rows show 3D dissections – for each set we show the 3D pieces on the left, and two shapes constructed from them on the right. In particular, the fourth example is a dissection between a polycube bunny model and a cuboid. The insets in this example show partial constructions.  

2.2 An example where a square is dissected into a triangle. Source image from Wikipedia [36].  

2.3 Any triangle can be dissected into a rectangle.  

2.4 Examples of several discrete lattice grids.  

2.5 An overview of our hierarchical optimization algorithm for dissecting two input figures: one is a 15×10 rectangle with an off-center hole, and the other is a 12×12 square. At each level, we show the steps being performed, and visualize the changes in one of the candidate solutions after each step. This example requires 3 pieces to dissect, which was found by our algorithm at the end of level 2.  

2.6 Comparing solutions computed with and without area-based term. In both examples (a) and (b), two solutions are shown which achieve the dissection with equal number of pieces. Note how the area-based term leads to results where the size of each piece is more balanced, which is usually more preferrable.  

2.7 An example of 3D shape dissection. The first input is a 4³ cube with a 2³ cavity at the center, and the second is a 7x4x2 cuboid. The solution contains 4 pieces shown on the right. The two sets of images on the left show the assembly of the pieces into each input shape.  

2.8 A 3D shape dissection that illustrates 3³ + 4³ + 5³ = 6³. The solution contains 6 pieces that are shown in Figure 2.1 example three.  

2.9 Examples of three-figure dissections. The top row shows a 6-piece dissection of a square, a rectangle with a cross hole, and a solid rectangle. The bottom row shows a 13-piece dissection of a square, the Chinese character for ‘person’, and a figure of a person.  

2.10 Three examples of two-figure dissections using the triangular lattice. The examples in (b) and (c) dissect an English letter with an object figure whose name starts with that letter.
2.11 An example where the algorithm found multiple solutions with equal number of pieces. The input are a $9 \times 9$ Serpenski carpet and a $8 \times 8$ square. Five selected solutions are shown, all of which are 7-piece solutions. We calculate the area variance for each. Smaller variance corresponds to a more uniform/balanced size, which is generally preferable for aesthetic reasons.

2.12 A comparison of our solutions with reference solutions shown in [44]. Some of these examples are visualizations of the Pythagorean triple numbers. The left column shows the input, the middle shows our solution, and the right shows the reference solution. For all examples we achieve the equal number of pieces with the reference.

3.1 Folding a car into a cube. Our system finds a collision-free folding sequence.

3.2 An illustration of my method. (a) We first find the best voxelization of the input shape. (b) Geometric neighbors define the connectivity graph with nodes as voxels and edges as potential hinge locations. (c) We turn the graph into a tree. Some edges are removed, and some edges are turned into rigid links. The rest are assigned a joint type and a folding angle. (d) Once we compute the locations of the joints and their angles, the shape can transform into a box.

3.3 (a) The original, uniform voxelization may contain some small parts. (b) A small offset is applied to each of the planes to minimize the voxelization energy. (c) I move the planes back to their original locations, which deforms the mesh parts in the voxels. (d) Each voxel is further divided into sub-voxels.

3.4 Illustration of the fullness function


3.6 Illustration of surface energy

3.7 Using physics to find a folding sequence. Using repulsive forces and some user interactions, A and B can be unfolded into A’ and B’. (The reverse is not easy to do with physics.) Then A’ and B’ are matched to each other using attractive forces. Once A’ and B’ coincide, we have a collision-free path from A to B and vice-versa.
3.8 (a) The user sees that the correct unfolding sequence is to rotate the purple voxel around 1 and then the pink voxel (which is the purple voxel’s parent) around 2. (b) But the corner (marked in yellow) on the purple voxel prevents rotation. (c) The user carves some edges of the purple voxel and continues unfolding. ................................. 57

3.9 Results physically manufactured using a 3D printer. ......................... 59

3.10 (d) The U-car demonstrates that the target does not need to be a box. (e) The dragon contains some struts due to the challenging geometry. (f) The 5×5×5 elephant is the largest example produced. ............... 60

4.1 A stool before and after optimization. The middle part is deformed in order to stand more force. ......................................................... 61

4.2 An illustration of a deformed tetrahedron. The vertex $x_0$ is deformed to $p_0$. ................................................................. 64

4.3 Possible Laplacian transformations. The left image shows the original shape. The middle image shows rotation, and the right image shows non-uniform scaling. ......................................................... 69

4.4 A comparison between our method and local thickening. The input shape is a utility hook. We use each of our method and local thickening to generate an output that can withstand the same maximum force load. Note how our result is better at preserving shape features. In addition, the volume of our result is 20% less than local thickening, thus our method is more efficient for 3D printing as well. ......................... 74

4.5 Without symmetry constraints, the resulting optimized shape becomes asymmetric. This can be addressed by incorporating symmetry constraints. ......................................................... 82

4.6 Without user-defined constraints, the resulting optimized shape in (b) has a top surface that is deformed. This can be addressed by user-defined constraints that keep the top surface fixed. ......................... 83

4.7 Block diagram of our optimization approach. ............................. 84

4.8 Without extrinsic distance, the optimized shape may deviate significantly from the original (b); without intrinsic distance, the result can contain bumpy surface/jaggy edges and lose surface details (c). Including both measures is important to ensure the result is geometrically similar to the original. ......................................................... 84
4.9 A simple test shape .......................................................... 90

4.10 Results of the simple test shape. Right: color coding of the von Mises stress. .......................................................... 90

4.11 A gallery of experimental results. The first row shows the original shapes and region/direction that external forces are applied. We find the maximum force each original shape can withstand, and apply our algorithm to optimize the shape until it can withstand 50%, 100%, and 200% additional force respectively. The results are shown in the next three rows. Each result contains a rendered image of the optimized shape, a silhouette image overlaying the optimized shape on the input to show the deformation, and the color encoding of the vertex displacement (i.e. how far the vertex has moved before and after optimization) as well as von Mises stress distribution over the surface.......................................................... 91

4.12 Photographs showing the Instron Universal Testing machine, which applies increasing axial force on the shape until it breaks. It can accurately measure the force applied on the testing shape, and the distance that the load cell has moved (head extension). We use the machine to perform physical validations. The picture on the right shows a close-up view of the object being tested. ....................... 94

4.13 Physics validation results for the twist-leg table and the four-leg stool models. In each graph, the x-axis is the head extension in millimeter (i.e. the distance that the load cell has moved); the y-axis is the measured force in kilonewton. The original shape is plotted in gray, and optimized shape plotted in purple. The original and optimized shapes are normalized before printing, so they have equal volume/weight. The point where the curve reaches the bottom is where it breaks under the corresponding force. ....................... 94

4.14 Using our SPH sampling method to simulate the Lloyd’s relaxation profile. This regular hexagonal distribution results from ensuring the convergence condition \( \sum_j \nabla W_{ij} = 0 \). ....................... 97

4.15 Using our SPH sampling method to simulate the CCVT profile. This distribution results from the convergence condition that \( \rho_i = \rho_0 \). ............ 99
4.16 Sampling the interior and boundary of a 2D bunny shape using our SPH-based method. The color visualizes the density value at each point during the iterations. Red means high density, blue means low density, and purple in between. The boundary samples are important as they can influence the interior samples. The rightmost image shows converged result. Note the distribution quality of the resulting samples.

4.17 Sampling the interior and boundary of a 2D bunny shape using our SPH-based method. The color visualizes the density value at each point, from high (red) to low (blue).

4.18 In a the boundary points are irregularly spaced after sampling. In b, the points are more relaxed after adding the correction force, but the sampling rate is much denser than the interior. In c both correction and cohesion forces are applied, and the resulting boundary samples are coherent with the interior samples.

4.19 Comparison of Poisson disk surface sampling ([27]) and our method.

4.20 Surface sampling of a bunny model. Fig. 4.20(a) visualizes the samples. Fig. 4.20(b) shows the Differential Domain Function (DDF) of the bunny surface samples (averaged over 10 runs) and Fig. 4.20(c) shows the DDF of the CCVT method [18], both calculated using the method of Wei and Wang [130].

4.21 In a and b we show the result of Fattal [43] and our method respectively.

4.22 Comparison of adaptive sampling of our method vs. [18] and [43]. The spatial density function is defined as the quadratic ramp shown on the top. Every example contains 1000 points, the percentages underneath indicate the density ratios for each quarto of the ramp.

4.23 If the smooth radius is smaller than the feature size b, the interior boundary will be correctly sampled. Otherwise the sampling on the interior boundary will be incorrect.

4.24 Effective Nyquist frequency and Oscillation changes as we increase the ‘density difference’ parameter.
4.25 Trade-off between Effective Nyquist Frequency and oscillation. The blue line shows the range that our SPH sampling method can achieve, by varying the ‘density difference’ parameter. We compare with other methods by plotting their trade-off in the same figure. Note that most of them can only generate one distribution profile, therefore are represented by single dots. The data source of these methods is from Heck et al. [52].

4.26 The surface of the bowl is remeshed with 5000 samples. With our method, points are implicitly attracted to regions of high curvature.

4.27 The bunny model is adaptively sampled based on a size function deduced using the medial axis [4], resulting in dense sampling of the ears, tail and feet, and sparse sampling of the body. The cutaway demonstrates that both the surface and internal volume is well adaptively sampled. The radius and color of particles is set according to the size function.
CHAPTER 1
INTRODUCTION

1.1 Motivation

Today we are witnessing a rapid growth in the 3D printing technology. Each year, manufacturers are developing and releasing printers with new features, support for a wide range of materials, more affordable price, higher printing quality, and larger capacities. The applications have also expanded from rapid prototyping and design to art, education, medical applications, food and fashion industries. Both online and retail services are increasingly available, allowing users to more easily and quickly convert virtual designs to physical 3D objects. In a way 3D printing is rapidly democratizing manufacturing, allowing users to easily produce customized 3D objects in office or at home.

The increasing availability of 3D printing also presents new research opportunities and challenges. Like traditional media such as 2D images and videos, one challenge is content creation — how to allow an average user to quickly and efficiently design 3D models for printing. Although 3D printing makes it easy to fabricate customized objects, the problem to design customized and functional objects still remain challenging for novice users. Besides, creating 3D content usually requires considerably more efforts and skills than creating 2D content, and 3D modeling software often incurs a high learning curve. The overarching goal of this thesis is to develop techniques to help inexperienced users create 3D content. Some unique challenges include how to ensure the resulting 3D models are feasible (such as in the case of 3D puzzle design), printable (i.e. manufacturable), meet required physical standards, and are optimized to reduce printing cost and time.
In this work, I present several aspects of 3D shape design and optimization for 3D printing. Design is one of the most direct ways to create contents for 3D printing. With sufficient skills, designers can create objects ranging from small tools such as screwdrivers and wrenches, to large-scale objects such as buildings and architecture. With modern computer technology, and also due to the increasing complexity of 3D models, computer-aided design has become more ubiquitous. Today many design tasks, ranging from tedious and labor intensive ones, to challenging and creative ones, are routinely solved using computational solutions. The first part of this thesis will focus on geometric puzzle design, which is a popular application of 3D printing for entertainment and mathematical art. Here the goal is to rapidly design customized geometric puzzles, which can then be printed as physical puzzles for users to enjoy or for geometry education. The challenge is to come up with efficient computational algorithms to quickly solve the puzzle design problem. In general, puzzle design can be thought of as a shape optimization problem – compute the best set of geometric pieces that can satisfy the given geometric constraints. I will begin the discussion with the classic problem of 2D geometric dissections, and discuss my work on converting it to a computational problem. I will then show how it can be extended to 3D dissections, and hinged dissections. On a high level, the user provides two desired input 3D models, and our algorithms will automatically compute the best set of puzzle pieces to satisfy the dissection constraints. Thus, without professional experience and skills in mathematical puzzles, the user can quickly design new, customized geometric puzzles with desired input shapes.

Another way to create 3D content is to start with existing 3D shapes and optimize / deform them to create new 3D shapes. One particular application is functional 3D shape design. For example, existing CAD software can help users quickly build digital models with the desired shape and aesthetics. While these models are useful for rendering, gaming, and virtual reality, they may not have the required physical strength when printed and placed into practical use. For example, a wrench, while shaped beautifully, may not be
Figure 1.1: Two dissection results from my algorithm. In each example, a set of puzzle pieces can be rearranged into construct both the left and right figures. The pieces are color-coded to indicate the solutions. Left: demonstration of the Pythagorean triple numbers $3^2 + 4^2 = 5^2$. Right: a heart figure that can be rearranged into a key figure. My algorithm computes each example with as few pieces as it can (i.e. minimum solution). The resulting solution can be used in applications such as recreational art and geometry education.

able to withstand the force required when tightening screws; a stool, while designed with great aesthetics, may not be able to withstand the necessary weight or force. With an optimization process, one can start from an existing 3D model that a designer created, and make it more suitable for functional use by improving its physical strength. So in the second part of the thesis, I will focus on the problem of physically-based shape optimization to help novice users quickly design functional shapes. Here the application is to use 3D printing to create functional tools and parts, such as wrenches, gears, cloth hangers, stools. The practical issue is that these tools or parts must withstand certain forces and stress. A casually designed 3D model may not meet the force or stress requirement and hence may break or collapse during use. The goal of my work is to apply optimization techniques to computationally refine 3D models in order to meet the required physical standard. For novice users, this is a particularly useful tool as it can significantly reduce the prototyping time and ensure the resulting 3D models meet the design specifications.

1.2 Design of Geometric Dissection Puzzles

The design of geometry dissection puzzles is a long-standing problem that has fascinated mathematicians over a long time. It is an excellent application for 3D printing,
because the availability of 3D printer allows users to easily create and manufacture customized puzzle designs. Traditionally, dissection puzzles have been designed by skilled mathematicians that have years of training and professional experience. The challenge is to find the minimum solution that can solve the dissection problem with as few pieces as possible (see below for details). To tackle this issue, in Chapter 2 I have proposed the first computational algorithm that uses an optimization-based approach to compute the minimum solution to the geometric dissection problem, and used it for creating a variety of new puzzles. This is the first attempt to solve the dissection problem using computational techniques. On a high level, the users will provide two preferred shapes as input, then the algorithm will automatically turn the input shapes into a minimum set of pieces that satisfy the dissection constraints. Using this algorithm, users can focus on providing the desired shapes, while the challenge of finding the dissection solution is offloaded to computers. A quick example of our puzzle is shown in Figure 1.1.
Mathematically, geometric dissection seeks to cut one figure into pieces which can be reassembled to construct one or multiple other figures. For a long time, geometric dissections have enjoyed great popularity in recreational math and puzzles [73, 44]. One of the ancient examples of a dissection puzzle was a graphical depiction of the Pythagorean theorem. Today, a popular dissection game is the Tangram puzzle [110], which uses 7 geometric pieces cut from a square to construct thousands of distinct shapes. Geometric dissection is also closely related to tiling and tessellation, both of which have numerous applications in computer graphics and computational geometry.

Early studies in this field focused on the question of whether any two shapes of the same area have a dissection solution, that is, if they can be cut into a finite number of congruent pieces [44] thus the same set of pieces can construct both shapes. It is long known that a dissection always exists for 2D polygons, due to the Bolyai-Gerwien theorem [78, 26, 47]. Although the theorem provided a general solution to find the dissection solution, the upper bound on the number of pieces is quite high. In practice, many dissections can be achieved with far fewer pieces. Figure 1.3 gives a simple example. The optimal dissection is unintuitive and cannot be constructed by the Bolyai-Gerwien theorem. Therefore much recent work has focused on the challenge of finding the optimal dissections using as few pieces as possible, and this has inspired extensive research in the mathematics and computation geometry literature [73, 35, 44, 38, 63, 7]. While many ingenious analytic solutions have been discovered for shapes such as equilateral triangles, squares and other regular polygons, finding the optimal solution for general shapes remains a difficult open research problem.

In Chapter 2, my goal is to seek an efficient computational algorithm for the geometric dissection problem, and my solver can be used to facilitate the creation of dissection puzzles. To do so, an optimization approach is employed here that operates in a discrete solution space. To make the problem tractable, I make the simplifying assumption that the input figures can be represented onto a discrete grid such as a square lattice. Each input
figure is rasterized into the lattice, and I provide a simple editing interface to modify the rasterized figure or create one from scratch. Following this step, the dissection problem can be reformulated into a cluster optimization problem. Specifically, the goal here is to partition each figure into the smallest number of clusters (pieces) such that there is a one-to-one and congruent matching between the two sets. Two pieces are considered to be congruent if they match exactly under isometric transformations, including translation, rotation, and flipping. As this is a combinatorial optimization problem, a brute-force solution is intractable even for small-scale problems. Therefore I have proposed a hierarchical clustering method that can efficiently find an optimal solution by iteratively minimizing an objective function. The main idea is to start with two clusters in each figure, search for the partitioning that gives the best matching score, then progressively insert more clusters at each subsequent level until a dissection is found. The matching score is defined using a distance metric that penalizes mismatches. During optimization, the search is prioritized towards directions that are more likely to reach a dissection. The algorithm can efficiently converge to a solution with a small number of pieces. Furthermore, I have found the solutions to be optimal for all test cases that we can verify optimality.

With the computational approach, the creation of puzzles can be extended in several ways. First, the square lattice can be replaced with a triangular lattice which can account for $45^\circ$ angled edges in the input figures. Other regular grids, such as the hexagonal lattice, are also possible. Second, the objective function can be modified to include an area-based term, which favors pieces with a more balanced size. This can help avoid solutions where some pieces are significantly larger than other pieces, which can reduce the playability of the puzzles. Third, I also show an extension of my algorithm to dissecting multiple input figures. A global refinement is proposed to simultaneously optimize all input figures, instead of a trivial approach that simply overlays the pairwise dissections. Finally, the algorithm is also extended to dissecting 3D shapes, thus creating 3D geometric puzzles.
Figure 1.2 shows several examples produced using my method and 3D printed into physical objects.

It should be noted that as the input is required to be discretized, my method is not meant to substitute the analytic approaches to many general dissection problems. Rather, our aim is to find an efficient computational solution, which provides a convenient tool for users to create a variety of different dissection puzzles.

1.3 Design of Box-shape Geometric Puzzles

Following the geometric dissection puzzle described in the previous subsection, the next puzzle design problem I present is to compute printable transforming box-shape puzzles. Here the goal is to partition an input 3D shape into pieces that can transform into a box shape while keeping all pieces connected (hinged) at all times. While conceptually this is similar to the classic dissection problem, it is a quite unique problem by itself. First, the transformed shape matches a box only on the exterior – internally it may be hollow or contains empty spaces. This is different from the goal of geometric dissections. Also, since all pieces must be connected at all times, this is similar to a hinged dissection, which is more challenging than the classic dissection problem where each piece can move independently.

This work is inspired by the observation that humans are fascinated by objects that have the ability to transform into different shapes. My interest is especially piqued when these shapes are dissimilar. Image-morphing and mesh-morphing have this type of appeal [132,
but it captivates us even more when we watch the process of transformation. This has recently been exploited in motion pictures such as Transformers [119]. In the virtual world, such transformations can take place in any arbitrary way without observing physical plausibility. In contrast, my work is focused on generating physical puzzles that can be 3D printed, and the puzzle needs to be physically plausible. This challenge is also the focus of study for many other recent works on the creation of 3D puzzles [75, 133, 111]. They captivate us arguably for a similar reason—the simple building blocks can be assembled and transformed into a complex final model.

In this part of the research, I tackle both of these challenges together, namely creating transformations of 3D shapes that are manufacturable. I focus on one specific type of shape transformation: folding 3D objects into a cube or a box-like shape (1.4). A cube is considered to be a special shape as it is highly symmetric and regular (one of the platonic polyhedra). Cubes and boxes are often seen as the most basic 3D shape that does not resemble any specific object. They can be stacked, stored and transported more easily, and used as “building blocks” for other shapes. My work presents a method to create a fabricated 3D object that can physically fold between the input 3D shape and a box. Unlike previous works in computer-assisted fabrication that create disjoint pieces [83, 80, 53, 106, 30], my method produces a single, connected object that can be folded. Along with the visual appeal and functional advantages of stacking and transporting, my technique allows for reduced printing times and cost, due to the compactness and regularity of the shape.

Given the input 3D shape and the target box dimensions, finding a physically achievable folding sequence is a challenge as it involves many sub-problems that are interdependent. The input shape needs to be segmented into parts, and these parts need to be connected in a pattern that can fold into two configurations—the source and the target shapes. Both the static configurations as well as the dynamic folding sequence need to be physically achievable. This means that parts should be able to fold, joints should be sturdy, and self intersections or interlocking should not occur at both configurations and each step of the
Figure 1.5: An illustration of my method. (a) We first find the best voxelization of the input shape. (b) Geometric neighbors define the connectivity graph with nodes as voxels and edges as potential hinge locations. (c) We turn the graph into a tree. Some edges are removed, and some edges are turned into rigid links. The rest are assigned a joint type and a folding angle. (d) Once we compute the locations of the joints and their angles, the shape can transform into a box.

Theoretically these problems can be shown to be very difficult. For instance, we examine the two subproblems of computing a segmentation with a connectivity structure (the joints), and finding physically achievable folding sequences for a given structure. For the first one, there exists an algorithm for placing joints given the common dissection between two shapes, but finding common dissection itself is an open problem in 3D [1]. In addition, this algorithm tends to cut the shape into a large number of tiny structures, which are implausible for actual 3D printing. For the second subproblem, one can prove it is PSPACE-complete (not less difficult than NP problems), by reducing it from the 2D linkage tree reconfiguration problem [13]. And it is also well known in the protein folding community that just finding the minimum energy state given a set of joints is NP-complete [22].

To make the search space tractable, and to find a plausible solution I make some underlying design choices. Instead of using arbitrary segmentation and arbitrary joint angles,
voxels are used as the folding primitives with a discrete set of joint angles between them. Hence, the segmentation problem here turns into a voxelization problem. Next, the connectivity structure for the voxels needs to be determined. Joints will be placed only between connected pieces that need to move during folding. Since the whole object must be connected, such a pattern forms a connectivity graph on the voxels. Connectivity loops in this graph are plausible and could potentially increase the stability of the static configuration. However, since they typically cause complex locking patterns in the folding sequence, I choose to constrain this graph to a tree structure. Each tree edge represents a connection between neighboring voxels. If these voxels must move relative to each other during the folding sequence, a joint must exist. Our problem is therefore to choose the location of these joints and then compute the angles so that the initial shape will fold into the target shape.

In some cases, instead of using a box directly as the target shape, we will use a template that can be easily folded into a box (see example on the right). Using such a template not only makes the search for solution easier but also reduces the printing time, since we can print the object in a compact, flattened state.

Even after limiting the scope to voxels, the size of the search space is still too large, and therefore we cannot hope to exhaustively search through all possible folding patterns. Finding a solution manually is possible only for small examples with a handful of pieces (e.g., cubebots [129]), but I want to be able to produce outputs with as many as 125 pieces. Thus I use simulated annealing [62] along with beam search [77] to search the space of solutions.

In general, I seek a solution that optimizes a number of objectives:

- **Geometric fit**: The folded object must match the target shape.
• **Compactness:** The space wasted in the folded shape must be minimized.

• **Fabricability:** All the joints and connectors must be printable. Small pieces must be avoided.

• **Foldability:** There must be a physically achievable sequence of moves to fold/unfold the shape with no intersections.

Trying to solve all of these at once imposes a major challenge. The key to my solution is the separation of the problem into three stages: defining the shape, finding the connectivity structure, and finding the folding sequence. In the first step, I search for a good voxelization pattern of the input 3D shape following the first three objectives above. Because the voxels in the input shape are packed, it is difficult to search for a solution that already maintains all the objectives. In the second step, I simultaneously build a connectivity tree between the voxels and search for a folding sequence that transforms the input shape into the target shape by following only the first objective above. This step only defines the connectivity structure of the object that can fit the source and target configurations. Only in the third step I follow the fourth objective and search for a non-intersecting folding sequence. However, instead of searching for a folding sequence from the source shape to the target, I utilize a physical simulator to unfold both configurations and match them. This provides a valid sequence of folding moves that will transform the object from the input shape to the target shape in a plausible manner.

### 1.4 Optimization for Functional Shapes Design

The geometric puzzle design problems I present in Chapters 2 and 3 allow users to easily create customized puzzles, without requiring the users to have professional skills and experience. In the next part of my thesis, I will move onto a different problem, which is an optimization-based method for improving existing 3D shapes. This allows novice users
(who may have little experience with mechanical engineering) to quickly design functional shapes that meet specific physical standards (e.g. maximum internal stress).

Fabrication of functional tools and parts is one of the major applications of 3D printing. Imagine that in the future, a lot of everyday physical objects do not need to be purchased any more. Instead, you can purchase the digital content online and send it to your 3D printer to manufacture the physical objects on demand. I believe it will not take long for this to become reality. Assume a user wants to create a 3D model, be it a utility tool or a piece of furniture, due to the lack of professional knowledge, the user’s design may not provide the required physical strength when printed. For example, a 3D printed bottle opener may break easily if the shape is poorly designed; a utility hook may break easily if the weight hanging on it exceeds a certain threshold. Although a novice user can quickly design a 3D prototype with desired shape and aesthetics, or simply go online (e.g. Thingiverse [117]) to download someone else’s design, there is no guarantee that the printed 3D shapes can function properly under the specified physical conditions. It would help tremendously if we can design software that help the users automatically optimize the 3D shapes such that they won’t fail in practice.

Professional designers may use their expertise in mechanical engineering to help design robust tools and parts. But for inexperienced users, they may lack of professional knowledge for their design. To lower the bar for 3D printing, it would be ideal to have a computer algorithm to automatically optimize the user’s design to ensure it meet the physical standard. One approach to this problem is shape optimization, which has been studied for decades in the mechanical engineering community [51] [103]. These algorithms are typically used to optimize the design parameters of specific classes of objects. However, existing methods almost all require parametric representations of the shape, which is not always available especially for complex shapes.

In this thesis, I study new optimization methods for functional shape design that is applicable to arbitrary input shapes. Chapter 4 discusses my solution to this problem. On
a high level, the workflow of the method is as follows: the user provides an initial design; then, we run physically-based analysis (e.g. calculate von Mises stress) to examine how the design may fail under significant external force; this is followed by a shape optimization step to deform the shape towards improved structural strength. By iteratively optimizing the 3D shape, the algorithm improves the physical property of the shape until it can withstand the required force without breaking.

The algorithm contains 3 components: physics simulation, shape similarity analysis, and optimization. The first component is the physics simulator, which analyzes the stress distribution under the user-specified external force. The goal of the algorithm is to optimize the shape globally until the maximum internal stress falls below the material’s threshold (e.g. the material will not yield under the force). The second component is the shape similarity analysis, which defines how different the optimized shape is from the original shape. Obviously improving the structural strength of the shape requires deformations. But it’s usually desirable to keep the resulting shape as similar as possible to the original shape that captures the user’s design. Therefore a shape similarity metric is necessary to preserve the shape of the object, and to minimize the modifications to the shape. During the optimization, the algorithm will try to preserve the similarity as much as possible, therefore this is a constrained optimization problem. The last component is the optimizer. Unlike previous work, one unique feature of my optimizer is that it integrates the physics simulator into the optimizer, instead of treating the physics simulator as a black box. As will become evident later, this makes it feasible to apply optimization directly on the mesh representation of the shape (which was previously believed to be infeasible). The optimizer tries to solve a large-scale, highly non-linear constrained optimization problem, and a generic optimizer would be inadequate for this purpose. Therefore I describe how to construct a problem specific optimizer tailored to this particular application.

An additional feature of my algorithm is the accommodation of user-specified constraints. This is often a desired feature for users, which allows them to impose customized con-
straints during the optimization process to keep certain aspects of the design intact. For example, the user may want a table model to have a specific height – while the optimizer can deform the shape as it needs, it will keep the height of the table unchanged. My algorithm allows such constraints to be easily included.

During my experiments, I also observed a technical issue caused by non-uniform sampling. Specifically, some shapes tend to fail with my algorithm, mainly due to problems with the mesh itself. The mesh that represents the shape needs to be tetrahedralized/hexahedralized uniformly for the physics simulator. But in general many surface meshes are represented as a set of triangles, and these triangles can have arbitrary sizes / aspect ratios. There is no general guideline to discretize such meshes into well-shaped tetrahedra/hexahedra with uniform size. So the surface triangles are usually very non-uniform especially for meshes with small features. Existing tetrahedralization algorithms need to include surface vertices into the volumetric mesh, thus the non-uniformly distributed vertices affect the quality of the tetrahedralization. Technically this inevitably results in near-zero Jacobians (unless if we discretize the mesh into a prohibitively high number of tetrahedra), which causes numerical stability issues. One way to solve this issue is to resample the mesh prior to tetrahedralization. The resampling algorithm should ideally give uniform samples and at the same time respect sharp features.

To ensure uniform tetrahedralization, I have studied a new method that uses blue noise samples for tetrahedralization. Specifically, my work uses smoothed hydrodynamics particles (SPH) method to simultaneously sample the surface as well as the interior of a 3D shape, resulting in uniform distribution. The sample points will then be used to generate a tetrahedral mesh for physics simulation. The method is described in Section 4.6. In my algorithm, sample points on mesh surface can be well aligned with sharp features of the input mesh. As a result, the reconstructed mesh can well preserve such features, as is shown in Fig. 1.7.
Figure 1.7: Comparison of surface reconstruction using Bilateral Sampling [31] vs our method. The surface of the bowl is remeshed with 5000 samples. Note how our method preserves sharp features.

To begin with, let me provide a short introduction to blue noise sampling. Blue noise sampling is a well-known technique useful in many graphics applications, such as image synthesis, physically-based simulation, non-photorealistic rendering, and geometry processing. It generates sample points that are stochastic yet evenly dispersed in the spatial domain. Fourier spectrum analysis shows that the spectral energy of such sample points is largely absent in the low-frequency (i.e. ‘red’ in terms of the visible spectrum) region, while evenly spread in the high-frequency (i.e. blue) region. This ‘blue noise’ property turns out to greatly benefit anti-aliasing, and the sample patterns are also visually pleasing, leading to its popularity in the aforementioned applications.

Beginning with the first dart throwing algorithm [37, 88], various methods have been proposed to generate samples with different blue noise patterns, such as Lloyd relaxation [74], Poisson disk sampling [81], Capacity Constrained Voronoi Tesselations (CCVT, [18]), kernel-density model [43], tiling [124]. The blue noise property of the above methods can be characterized by its anti-aliasing and noise property [52]. Some methods have better
anti-aliasing property, such as Poisson disk [81], while some performs better in terms of reducing noise, such as Lloyd relaxation [74]. There is always a trade-off between those two properties, and choosing this trade-off would become an important factor when selecting blue noise sample techniques for a given application.

The above methods are focused on generating blue noise samples with uniform distribution. For our tetrahedralization application, feature preserving is another desired property. Specifically, our samples are preferred to align with the edges on the surface, so that the sharp features can be reserved. Recently, Bilateral Blue Noise Sampling [31] has been presented as a sampling method for the feature-preserving blue noise sampling. Its main principle is to incorporate surface normals into the distance metric for dart throwing. Nevertheless, their method cannot simultaneously handle volumetric and surface sampling, which makes it infeasible for tetrahedralization. Also, users have no control over the the spectral distribution properties in this method.

Thus, in the later part of Chapter 4, I discuss our new blue noise sampling method inspired by the Smoothed Particle Hydrodynamics (SPH) method. Our method provide controllable distribution property by tuning a single parameter, which makes it suitable for different applications. For the application of tetrahedralization, our method samples both the volume and the surface simultaneously. The surface samples align nicely with surface edges, as a result surface features can be preserved nicely. Thus, our method provides tunable blue noise pattern as well as feature preserving property, which makes it suitable for improving the tetrahedralization quality.

In summary, in Chapter 4, I discuss the optimization approach for functional shapes design. Specifically, it can help novice user to strengthen an existing design to guarantee that it can meet the physics standard. The later part discusses an SPH-based algorithm, which aims to solve the non-uniform tetrahedralization problem that affects the performance of our shape optimizer. With this method, I hope to further lower the bar of 3D printing so
that even users with little mechanical engineering expertise can design practical tools and parts with required physical strength.

1.5 Contributions

My dissertation is focused on studying several selected aspects of shape design and optimization problems for 3D printing. The overarching goal is to provide automated algorithms to help inexperienced users to easily create 3D content for printing. The dissertation consists of the following technical contributions:

- The first computational solution to classic geometric dissection puzzles in discrete settings. The algorithm can be used to create novel, customized geometric puzzles, both in 2D and in 3D, for recreational and educational purposes.

- An efficient algorithm for folding any given 3D model into a box-like shape. The algorithm finds the optimal splits of the input model into connected pieces, and searches for physically plausible folding paths. It can be used to create novel, customized folding 3D puzzles.

- A optimization-based algorithm to strengthen a given 3D model so that it can withstand user-specified external force. The algorithm applies continuous optimization integrated into a physics simulator on an input shape, and the resulting shape improves upon the input shape in terms of structural strength. It can additionally accommodate user-specified constraints. The algorithm can assist users in designing functional tools and parts.

- A SPH sampling algorithm which can create uniform sample points with blue noise properties in 3D volume. This algorithm can create feature-preserving blue noise samples, which can serve as a great starting point for tetrahedralization – a critical step for physics simulation.
CHAPTER 2

COMPUTATIONAL ALGORITHM FOR CREATING GEOMETRIC DISSECTION PUZZLES

In this chapter I will discuss the computational geometric dissection problem for customized puzzle design. As I described earlier, geometric dissection is a popular way to create puzzles. Given two 2D figures of equal area, a dissection seeks to partition one figure into pieces which can be reassembled to construct the other figure. In this chapter I present a computational method for solving this problem in discrete settings. Examples of our results are shown in Figure 2.1. In the top two rows in Figure 2.1, I show an example where the dissection pieces can be rearranged into three different simple geometries. In the bottom two rows I extend my algorithm to dissect discretized 3D figures. As is demonstrated in Figure 2.1, those examples can be easily fabricated with 3D printers. My algorithm can automatically design such puzzles for fabrication and geometry education purpose.

This chapter will be organized as followed. In Section 2.1 I will briefly go over the background of geometric dissection and its related work. Our algorithm is discussed in Section 2.2. Firstly an overview is given in Section 2.2.1. My method starts by representing the input figures onto a discrete grid, such as a square or triangular lattice. The goal is then to partition both figures into the smallest number of clusters (pieces) such that there is a one-to-one and congruent matching between the two sets. The first step in solving such problem is to define the matching, which is discussed in Section 2.2.2. We also define a matching score to quantitative evaluate the match between any two clusters. With the matching score, we can consider searching for dissections which minimize such score. The most intuitive solution would be a brute-force approach. Unfortunately such approach is
Figure 2.1: Four example sets of dissection puzzles created using our algorithm. The top two rows show 2D dissections – for each set we show three constructed figures. The bottom two rows show 3D dissections – for each set we show the 3D pieces on the left, and two shapes constructed from them on the right. In particular, the fourth example is a dissection between a polycube bunny model and a cuboid. The inlets in this example show partial constructions.
intractable since the solution space is exponentially large in terms of input figure. Thus, in Section 2.2.3 and 2.2.4, I propose a hierarchical clustering method that can efficiently find near-optimal solutions by iteratively minimizing an objective function. In such framework, each level proposes promising candidate solutions, which will then serve as good starting point for next level. To count for the aesthetic aspect of the dissection, in Sec. 2.2.5 I also modify the objective function to include an area-based term, which directs the solution towards pieces with more balanced sizes. I also show extensions of our algorithm for dissecting figures in other discrete domain, including triangular lattice (Section 2.3.1) 3D shapes of equal volume (Section 2.3.2) and multiple-figure dissection (Sec. 2.3.3). The results of my algorithm is demonstrated in Section 2.5. As we can see, they can be either physically built with Lego bricks or fabricated with 3D printing. The fabricated puzzles are interesting to play with and non-trivial to solve.

2.1 Background and Related Work

![Image of geometric dissection]

Figure 2.2: An example where a square is dissected into a triangle. Source image from Wikipedia [36].

**Geometric Dissections.** Fig. 2.2 shows a simple example of geometry dissection between a square and a triangle. As we can see, the square is dissected into 4 pieces, which can then be rearranged into a triangle. This kind of problems have a rich history, originating from the explorations of geometry by the ancient Greeks [12]. In Arabic-Islamic mathematics and art, dissection figures are frequently used to construct intriguing patterns.
ornamenting architectural moments [99]. Dissection figures also provide a popular way to create puzzles and games. The Tangram [110], which is a dissection puzzle invented in ancient China, consists of 7 pieces cut from a square and then rearranged to form a repertoire of other shapes.

When seeing such dissection puzzles, one may easily raise two mathematical questions: (i) does there exist a dissection between any two shapes; (ii) if so, how to find the dissection with the minimum number of pieces. The first question has a positive answer for 2D polygons with equal area. The proof can be found in several works [78, 126, 26, 47], and is commonly known as Bolyai-Gerwien theorem. The sketch of the proof is as followed. Firstly suppose we have a polygon with area $A$. Since a polygon can always be dissected into a finite number of triangles, we can then focus on each triangle. A triangle can always be dissected into a rectangle, as is shown in Figure 2.3. We do it for each triangle, and we can dissect the polygon into a set of rectangles. Each of those rectangles can then be dissected into a rectangle with unit width. By concatenating those rectangles together we will have a rectangle with unit width and the height of $A$. Till now, we dissect a polygon with area $A$ into a rectangle with unit width and the height of $A$. This rectangle can serve as the intermediate dissection between any two polygons with the same area. By imposing one intermediate dissection on another, we will have a common dissection between two polygons.

![Figure 2.3: Any triangle can be dissected into a rectangle.](image)

A stronger theorem also holds that there exists hinged dissections between any two 2D polygon with equal area [2]. A hinged dissection is a dissection that all dissected pieces
must be connected by hinges. Readers may turn to the work by Abbot et al. [2] for more details.

We have already answered the first question whether there exists a dissection between any two shapes. The second question about how to compute a minimum dissection is a more challenging problem, and has no universal answer. Thus, people have been focused on finding optimal dissections that use the fewest number of pieces. For example, Cohen [35] studied economical triangle to square dissections; Kranakis et al. [63] studied the asymptotic number of pieces to dissect a regular $m$-gon into a regular $n$-gon; Akiyama et al. [7] studied the optimality of a dissection method for turning a square into $n$ smaller squares; Czyzowicz et al. studied the number of pieces to dissect a rational rectangle into a square [38], and under the additional constraints of glass cuts [39]. In addition, the popularity of such problems is culminated in seminal books such as [73, 44]. Despite extensive research, finding the minimum dissection solution has so far only been possible for a few special cases, while the general cases remain an open research problem. Our work is first to present a computational algorithm to solve a general dissection problem in discrete domain.

**Tiling.** A closely related subject to geometric dissections is tiling [49], the basic form of which is to seek a collection of figures that can fill the plane infinitely with no overlaps or gaps. The use of tiling is ubiquitous in the design of patterns for architectural ornaments, mosaics, fabrics, carpets, and wallpapers. It is also seen throughout the history of art, especially in the drawings of M.C. Escher. In computer graphics, Kaplan and Salesin [59] presented a technique called ‘escherization’, which can approximate any closed figure on the plane into a tileable shape, simulating Escher-style drawings. A number of well-known tiling patterns, such as Penrose tiling, polyomino tiling, Wang tiles have also been cleverly applied in graphics, especially for blue noise sampling [98, 97] and texture synthesis [34, 45, 64]. An excellent introduction and survey of tile-based methods in computer graphics can be found in [66].
Tiling can also be used to create puzzles. Lagae and Dutré [65] have shown that the tile packing results can be used to create interesting jigsaw puzzles. Another relevant work is a method for creating 3D polyomino puzzles presented by [75]. Their method aims to find a set of polyomino pieces that can tile a given parameterized surface, and they designed clever interlocks to make the puzzles physically realizable. Generally, tile-based puzzles study how to use a predefined set of pieces to cover a given shape; in contrast, geometric dissection puzzles study how to solve for a set of pieces that can simultaneously construct two or more shapes. Thus their solution methods are considerably different.

**Recreational Math and Art.** Our work relates to a number of topics in computer graphics that are targeted towards recreational math and art, such as 3D Burr puzzles [134], ASCII art [136], paper popup [71, 72], camouflage images [33], shadow art [90], 3D polyomino puzzles [75], maze construction [135], papercraft models [87], jigsaw image mosaics [61]. Solutions to many of them involve solving a complex optimization problem. For example, Chu et al. [33] used a multi-label graph cut algorithm to solve an pixel labeling problem. In general, our formulation for geometric dissections can be viewed as a label assignment problem (the label being the index of a piece). However, we haven’t found any existing solution that can directly benefit our case. This is mainly because unlike in image domains, our objective function cannot be defined using a local coherence metric, thus an algorithm such as graph-cuts is not applicable.
2.2 Algorithms and Implementation

2.2.1 Assumptions and Overview

Given two input figures $A$ and $B$ of equal area, our goal is to find the minimum set of pieces to dissect $A$ and $B$. To formulate it as an optimization problem, we require both input figures to be represented onto a discrete grid. The simplest choice is a square lattice as shown in Figure 2.4(a), which is naturally suitable for representing rectilinear polygons. For other shapes, such as discs, we rasterize them into the grid, resulting in approximated shapes. Note that for the purpose of creating puzzles, exact representation of the input is not necessary. At sufficient grid resolution, the discretization typically produces acceptable shape approximations. Note that after discretization, the area (number of pixels) covered by each figure must remain the same. This can be ensured either by the design of the input figures, or by using a graphical interface (see Section 2.4) to touch up the rasterized figures. In the following, we use symbols $A$ and $B$ to denote the two rasterized figures of equal area.

Given the input, we formulate the dissection into a cluster optimization problem. Specifically, our goal is to partition each figure into the smallest number of clusters (each cluster being a connected piece) such that there is a one-to-one and congruent matching between the two sets of clusters. Here congruency refers to two pieces that match exactly under isometric transformations, including translation, rotation, and flipping. Since the solution space is discrete, the possible transformations are also discrete. For example, on a square lattice with the grid size 1, all translations must be of integer values, and there are only 4 possible rotations: $0^\circ$, $90^\circ$, $180^\circ$, and $270^\circ$. Thus excluding translation, two congruent pieces must match under the 8 different combinations of rotation and flipping.

Generally, solving such a clustering problem requires combinatorial search, which would impose a very large solution space. As the dissection requires the solution pieces to fit exactly with each other in both input figures, leaving no holes or overlaps, standard fitting or clustering algorithms are unlikely to lead to valid results. To efficiently solve the problem, we introduce a hierarchical clustering algorithm that progressively minimizes
an objective function until a solution is found. We start the search from a random initial
c Condition, and apply refinement steps to iteratively reduce the objective function value. We
use random exploration to keep the algorithm from getting stuck in local minima. Below
we will first describe our algorithm for dissecting two input 2D figures defined on a square
lattice, then describe its extensions to the triangular lattice, the dissection of multiple fig-
ures, and finally the dissection of 3D shapes. Figure 2.5 provides a graphical overview of
the algorithm.

2.2.2 Dissecting Two Figures on a Square Lattice

Distance metric. Given two pieces on each figure: \( a \subset A, b \subset B \), we define a distance
metric \( D \) that measures the bidirectional mismatches between them under the best possible
alignment:

\[
D(a, b) = \min_{T_{a,b}} \left( \left\| \{ p \mid p \in a \text{ and } (T_{a,b} \times p) \notin b \} \right\| 
+ \left\| \{ p \mid p \in b \text{ and } (T_{a,b}^{-1} \times p) \notin a \} \right\| \right)
\]

(2.1)

where \( T_{a,b} \) is an isometric transformation from piece \( a \) to \( b \), \( T_{a,b}^{-1} \) is the reverse trans-
formation, \( p \) counts the number of pixels that in one piece but not the other (i.e. it measures
bidirectional mismatches). As \( D \) measures the minimum mismatches under all possible
\( T_{a,b} \), it will be 0 if the two pieces are congruent.

To simplify the calculation of \( D \), we first set the translation to align the centers of \( a \)
and \( b \) together, then simply search among the 8 combinations of rotation and flipping to
obtain \( D \). While this does not consider other possible translations, we found it to work well
in practice, and it preserves the crucial property that congruent pieces must result in zero
distance. Note that if the center of a piece does not lie exactly on a grid point, we need to
align it to the 4 nearby grid points and calculate \( D \) for each; the smallest among them is
returned as the distance value.
Matching. Next, assume the two figures $A$ and $B$ have both been partitioned into $k$ clusters $\{a_i\}$ and $\{b_j\}$, we need to match the elements in $\{a_i\}$ to those in $\{b_j\}$ such that the sum of distance between every matched pair is minimized. We call this a matching between the two sets, denoted as $M$. Mathematically,

$$M = \text{arg min}_{m \in \{\{a_i\} \to \{b_j\}\}} \sum_{(a_i, b_j) \in m} D(a_i, b_j)$$

(2.2)

where $\{a_i\} \to \{b_j\}$ denotes a bijection from $\{a_i\}$ to $\{b_j\}$. Basically we are seeking among all possible bijections the one that gives rise to the minimum total distance. This is known as the assignment problem in graph theory, which is well-studied and can be solved by a maximum weighted bipartite matching. Specifically, we create a weighted bipartite graph between the two sets $\{a_i\}$ and $\{b_j\}$: every element $a_i$ in $\{a_i\}$ is connected to every element $b_j$ in $\{b_j\}$ by an edge, whose weight is equal to the distance $D$ between the two elements. The goal is to find a bijection whose total edge weight is minimal. A common solution is based on a modified shortest path search, for which we use an optimized Bellman-Ford algorithm [131]. It guarantees to yield the best matching in $O(k^3)$ time, where $k$ is the number of clusters.

We call the total pair distance under $M$ the matching score, denoted as $E_M$. In other words, $E_M = \sum_{(a_i, b_j) \in M} D(a_i, b_j)$. Note that $E_M = 0$ if $M$ is a dissection solution. Thus the smaller $E_M$ is, the closer we are to reach a dissection solution.

Objective function. Since the minimum number of pieces to achieve a dissection is unknown in advance, we propose a hierarchical approach that solves the problem in multiple levels. Each level $\ell$ partitions the two input figures into $\ell + 1$ clusters, and outputs a set of best candidates at that level. The basic definition of such an objective function is simply the matching score $E_M$. Specifically, let’s denote with $C_k = (\{a_i\}_k, \{b_j\}_k)$ a candidate solution where $\{a_i\}_k$ and $\{b_j\}_k$ are two given $k$-piece clusterings of $A$ and $B$ respectively; then the objective function $E_k(C)$ is:
At the end of each level $\ell$, we select a set $(N_b)$ of the best candidate solutions $\{S_\ell\}$ which give the smallest values according to Eq 2.3, and use the set for initialization in the next level. The algorithm will terminate when a solution is found such that $E(S_\ell) = 0$.

In the following we will describe our algorithms for the first and each subsequent level. Refer to Figure 2.5 for a graphical illustration.

2.2.3 Level 1 Optimization

**Seeding.** At the first level, our goal is to compute the best set of 2-piece clusterings to approximate the dissection. To begin, we split $A$ into two random clusters $a_1$ and $a_2$. This is done by starting from two random seeds, and growing each into a cluster using flood-fill. While we could also use other methods to grow the clusters, the flood-fill guarantees that each cluster is a connected component. We do the same for $B$, resulting in two random clusters $b_1$ and $b_2$. 

\[ E(C_k) = E_M(\{a_i\}_k, \{b_j\}_k) \]  

(2.3)
Compute matching. Now we have the initial sets of clusters \( \{a_i\}_2 \) and \( \{b_j\}_2 \), we can invoke Equation 2.2 to compute the matching \( M \) between them. In Figure 2.5 we use the same color to indicate a matched pair. Note that there is no particular ordering of the clusters, so the colors may flip depending on the output of the matching algorithm.

Forward copy-paste. Our next step is to refine the clusters. As the solution space is very large, randomly modifying each cluster by itself is unlikely to result in a better matching score. Therefore we introduce a more explicit approach that copies and pastes a cluster \( a_i \) to its matched cluster \( b_j \), in order to force their shapes to become similar. This is called a forward copy-paste. To do so, we apply the transformation which yields the distance between \( a_i \) and \( b_j \) (Equation 2.1) to \( a_i \), and pastes the result to \( B \). Note that if the two matched clusters are not congruent yet, the paste may overwrite neighbor pixels that belong to other clusters. This is allowed, but we randomize the ordering of clusters for copy-paste in order to avoid bias. Pixels pasted outside the boundary of a figure are ignored.

Following the above step, some pixels in \( B \) may have received no pasted pixels from \( A \), thus they become holes. We use a random flood-fill to eliminate the holes. Specifically, we randomly select already pasted pixels and grow them outward to fill the hole.

Random label switching. As mentioned above, during copy-paste, some clusters may overlap with each other, resulting in conflicts. Therefore our next step is to reduce such conflicts by modifying the cluster assignments for some pixels at the boundary of two clusters. To do so, we first recompute the matching between the current two sets of clusters, then simulate a copy-paste in the backward direction, i.e. from \( B \) to \( A \). During this process we record the pixels that would have overlapped after pasting. For each such pixel \( x \), we randomly relabel it to the cluster of one of its four neighboring pixels. This is called random label switching. Note that if \( x \) is surrounded completely by pixels of its own cluster, its label will remain the same. Thus only pixels on the boundary of a cluster can potentially be switched to a different label.
Intuitively, the motivation of the forward copy-paste is to encourage the clusters in $B$ to be shaped similarly to $A$, and the motivation of the random label switching is to modify the cluster boundaries in $B$ to reduce cluster conflicts/overlaps. The two steps combined is called a **forward refinement** step.

**Backward refinement.** The backward refinement performs exactly the same steps as the forward refinement, except in the reverse direction (i.e. a copy-paste from $B$ to $A$, followed by a random labeling switching in $A$). At this point, we have completed one iteration of back-and-forth refinement.

**Convergence.** We repeat the back-and-forth refinement iteration for $\times$ times (the default value of $\times$ is 100). This typically reaches convergence very quickly, upon which we obtain a candidate solution $C_2$, whose associated objective function value is $E(C_2)$.

**Random seed exploration.** The refinement process can be seen as a way to find local minimum from the initial seeds. Thus small changes to the initial seeds do not significantly affect the converged result. In order to seek global minimum, we apply random exploration, where we re-compute the the candidate solution $N$ times (the default value of $N$ is 400), each time with a different set of initial seeds. After random exploration, the best $N_b = 30$ candidate solutions (i.e. those with the smallest objective function values) are selected and output as the level 1 final results, denoted as $\{S_1\}$.

At this point, if there exists a candidate solution whose matching score is 0, we have reached a perfect dissection. Otherwise we continue with subsequent levels. The top portion of Figure 2.5 illustrates all steps in level 1. Note how the candidate solution refines following each step. The red outlines on some pixels indicate unmatched pixels between a pair of clusters which are not congruent yet.

**2.2.4 Level $\ell$ Optimization**

In each subsequent level $\ell$, we start from one of the best candidate solutions $S_\ell$ from the last level. Our goal is to insert a new cluster to $S_\ell$, and then search for the best $(\ell + 1)$-piece
approximation using the same back-and-forth refinement process as in level 1. Intuitively, as the output of the previous level are some of the closest $\ell$-piece approximations to the dissection, they serve as excellent starting points for the new level.

The main difference between level $\ell$ and level 1 is in creating the initial clusters. Note that we cannot use completely random seed initialization as in level 1, because doing so will completely abandon the results discovered in previous levels, and hence will not reduce the problem complexity. Instead, we introduce two heuristics to create initial clusters by exploiting the previous results, and we consider them both during the random exploration step.

**Splitting an existing cluster.** In the first heuristic, we select a pair of pieces $\{a_i, b_j\}$ from $S_\ell$ that has the largest (worst) matching score, and split each into two sub-clusters. We refer to the pair as the parent clusters. The splitting introduces an additional cluster for each figure; the remaining clusters which were not split remain unchanged for now. Next we need to decide how to perform the split. A straightforward way is random split, but as the parent clusters are not well matched, a random split can create difficulties for convergence in subsequent steps. Therefore we need to optimize the splitting to create better matched sub-clusters to begin with.

It turns out that we can optimize the splitting by using the same approach as level 1 optimization. To do so, we treat the parent clusters $a_i$ and $b_j$ as two input figures, and apply level 1 optimization to obtain the best 2-piece dissection between them. We have found this approach to work well in practice, creating sub-clusters that are matched as well as it can. Experiments show that this can significantly improve the quality of the subsequent refinement results.

**Creating new clusters from mismatched pixels.** Our second heuristic is to create a new cluster from the currently mismatched pixels. For example, assume $\{a_i, b_j\}$ are a matched pair but not yet congruent, then transforming $a_i$ to $b_j$ will result in some pixels that are not contained in $b_j$. These pixels will be marked in $a_i$ as mismatched pixels.
In Figure 2.5 the mismatched pixels are indicated with a red outline. After marking all mismatched pixels in $A$ and $B$, we randomly select a seed from them and perform a flood fill to grow the seed into a cluster, which then becomes a new cluster to be inserted to the current level.

**Comparing the two heuristics.** The rationale behind the first heuristic is that priority should be given to splitting the worst matched pair, as this is most likely to result in reduced matching score. The rationale behind the second heuristic is that when a candidate solution is very close to reaching a dissection, priority should be given to the few pixels that remain unmatched. In practice, we account for both of them during our random exploration: among the $N$ random tries, 75% will use the first heuristic to initialize the sub-clusters, and 25% will use the second heuristic. This way we can combine the advantages of both.

**Global refinement and random exploration.** Once the sub-clusters are created, we perform the same back-and-forth refinement process as in level 1. Now all clusters will participate in the refinement, therefore we call this step global refinement. Upon convergence, we obtain a candidate solution $C_{\ell+1}$.

In addition, we perform random exploration for $N$ times similarly to level 1, the goal of which is to seek global minimum. Each exploration starts from a randomly selected best candidate $S_{\ell}$ from the previous level, applies one of the two heuristics to insert a new cluster, and computes refinement. Again, after random exploration, the best $N_b$ candidate solutions are output as the final results $\{S_{\ell}\}$ of level $\ell$. Figure 2.5 shows an example of level 2 optimization. For this example, our algorithm discovered a perfect dissection at the end of level 2, thus the program terminates with a 3-piece dissection.

### 2.2.5 Area-Based Term

So far we have described a computational algorithm for finding the minimum dissection of two figures. However, there is no constraint on the size or shape of the resulting pieces. Thus a solution may contain pieces that are significantly larger than others. This is often
Figure 2.6: Comparing solutions computed with and without area-based term. In both examples (a) and (b), two solutions are shown which achieve the dissection with equal number of pieces. Note how the area-based term leads to results where the size of each piece is more balanced, which is usually more preferrable.

undesirable, both for aesthetic reasons and for reducing the difficulty of the puzzles (since large pieces are easier to identify and place on a target figure). For better control of the solution, we introduce an area-based term into our objective function in order to favor a solution where the size of each piece is more balanced. To do so, we modify Equation 2.3 to include the area-based term $E_{\alpha}$:

$$E = E_M(\{a_i\}_k, \{b_j\}_k) + \lambda \cdot [E_{\alpha}(\{a_i\}_k) + E_{\alpha}(\{b_j\}_k)]$$

(2.4)

where $\lambda$ is a weight adjusting the relative importance of the two terms. Here $E_{\alpha}$ is the total area penalty. It is defined by summing up the area penalty $\alpha(a_i)$ of each piece, which is calculated as:

$$\alpha(a_i) = \begin{cases} 
A(a_i)/\bar{A} - 1 & \text{if } A(a_i) > 2\bar{A} \\
\bar{A}/A(a_i) - 1 & \text{if } A(a_i) < \bar{A}/2 \\
0 & \text{otherwise}
\end{cases}$$

(2.5)
In the above equation, \( A \) denotes the area of a piece, and \( \bar{A} \) denotes the average area (i.e. the total area divided by the number of pieces). Essentially \( \alpha(a_i) \) penalizes a piece if it is either more than twice the average area, or less than half of the average area; otherwise we consider a piece to be within the normal range of size variations and assign a zero penalty.

Figure 2.6 shows an example comparing solutions computed with and without applying the area-based term. Note that while both solutions achieved equal number of pieces, enabling the area penalty leads to pieces of a more balanced size, which is often preferable. In addition, more uniformly sized pieces also tend to be symmetric with each other, which is a desirable property.

Note that the preference towards balanced area and the preference towards smaller number of pieces are often conflicting goals. For example, if the area weight factor \( \lambda \) is set too large, the solution will be heavily biased towards area uniformity, and will deviate from the goal of seeking the smallest number of pieces. To address this issue, we gradually decrease \( \lambda \) as the level \( \ell \) increases. This will reduce the effect of area penalty over levels, encouraging the solver to focus more on finding the minimum solution as the level increases. Our current implementation sets \( \lambda = \frac{1}{2} 0.8^{\ell-1} \).

**Avoiding split pieces.** Another improvement we made to the objective function is to include a term that penalizes split pieces. A split piece is one that contains disconnected components. While these components transform together in the same way, they are not physically implementable. Thus we simply add a large penalty to such pieces in order to eliminate them during best candidate selection. Note that we do not actively prevent them because there are cases where split pieces are temporarily unavoidable, such as during the first several levels of processing when the input figures themselves are fragmented (see Figure 2.12).
2.3 Extensions

2.3.1 Extension to the Triangular Lattice

 Besides using a square lattice, our method can also be extended to other lattices including the ones shown in Figure 2.4 (b,c,d). Currently we have implemented the right triangular lattice shown in (b), which is constructed by splitting each grid on a square lattice to four isosceles triangles along the diagonals. Using this lattice, we can represent input figures with both rectilinear edges as well as 45° angled edges. This usually makes the discrete representation more expressive. Figure 2.10 shows several examples.

 With the triangular lattice, our algorithms remain almost the same, because the possible transformations, including translation, rotation, and flipping, remain the same with a square lattice. The main difference is that a triangle pixel has three neighbors (those connected to it along the three edges) while a square pixel has four.

2.3.2 Extension to 3D Shape Dissection

 We can also extend our algorithm to dissecting 3D shapes that are represented onto a cubic voxel grid. In this case, each voxel has six neighbors, and the transformation of each piece considers 24 different 3D rotations. However, unlike 2D, a piece is not allowed to
be mirrored (which is analogous to flipping in 2D), because in general mirroring is not physically plausible in 3D. The area-based term is now modified to a volume-based term. Several examples of 3D dissection are shown in Figure 2.1, 2.7 and 2.8. Note that our implementation currently does not consider how the pieces can be locked together to form a stable 3D structure. Therefore it is possible that a piece may have no supporting pieces beneath it, thus the structure is not physically stable. Although we have not encountered this issue so far, it remains a future direction of research.

2.3.3 Dissecting Multiple Figures

Finally, we present an extension of our algorithm to simultaneously dissecting multiple figures. Note that a trivial approach is to simply overlay the pairwise dissection solutions, and output the intersections of all pieces. Unfortunately this will produce a large number of pieces that are overly fragmented – the upper bound is exponential with respect to the pairwise dissection results.

Here we achieve multi-figure dissection by adapting our optimization based algorithm. From Figure 2.5 we can see that the primary steps at each level of the algorithm consist of 1) cluster initialization and 2) cluster refinement. Below we discuss how these two steps are modified for a multi-figure setting respectively. The matching $M$ is still computed from a bipartite graph between a pair of figures. It is possible to redefine $M$ based on the complete $k$-partite graph among all $k$ figures, but computing the maximum weighted matching for such a graph is known to be an NP-hard problem.

**Multi-figure cluster initialization.** At level 1, the initial two clusters for each figure are created in the same way as before, i.e. each figure independently creates two random clusters using a flood-fill on random seeds. At each subsequent level, the initial clusters are computed using pairwise sub-cluster refinement. Specifically, we first pick a random figure as the pivot figure. Without loss of generality, let’s assume the pivot figure is $A$. Next, we select a piece from $A$ that has the worst total matching scores with all other figures,
and split it as well as its matched pieces in all other figures into two sub-clusters. These sub-clusters then need to be refined, for which we use the same back-and-force process as before, except that we now perform one iteration of refinement at a time, between the pivot figure and other figures in a round-robin fashion.

**Multi-figure cluster refinement.** As before, during global refinement, our goal is to modify the clusters in order to achieve improved matching result. To do so, we again select one figure as the pivot figure, then perform the matching and copy-paste from the pivot figure to all other figures. Next, we loop over all figure to perform random label switching. Here the candidates for label switching in a given figure is the set of all pixels that have at least one mismatch with any other figure. Once this is done, we proceed to the next figure as the pivot. Thus the original forward vs. backward refinement in the two-figure setting is now generalized to the multi-figure setting, where each figure will be used as the pivot once to perform a forward refinement with other figures.

Figure 2.9 shows two examples of three-figure dissection results. Note that these results achieve perfect dissections between all three figures. Since the computation for multi-figure dissection is more expensive, the running time is considerably longer than before.

### 2.4 Implementation Details

**Algorithm implementation.** A 2D figure is loaded from a binary image and stored as a 2D array. We represent a piece using the STL’s *set* data structure. The matching \( M \) between two clusters needs to be evaluated frequently, thus we use an optimized Bellman-Ford algorithm to quickly compute it. It is stored as a bidirectional list together with the transformations defined for each pair of pieces. We store the triangular lattice using a 2D array as well, where each array element stores the four triangle pixels, each at a different orientation. A 3D shape is loaded from a binary image representing each slice of the shape, and is stored as slices of 2D arrays. Finally, we parallelize the random exploration step
Figure 2.9: Examples of three-figure dissections. The top row shows a 6-piece dissection of a square, a rectangle with a cross hole, and a solid rectangle. The bottom row shows a 13-piece dissection of a square, the Chinese character for ‘person’, and a figure of a person.

with multiple threads, since each exploration is independently computed. This allows us to achieve linear speedup using a multicore CPU.

**Figure editing interface.** As our method requires the two input figures to contain equal number of pixels (or equal number of voxels in 3D), we implemented a simple user interface to assist the editing of input figures if necessary. For a user-provided figure, we first rasterize it onto the lattice grid, then allow the user to directly edit each pixel individually. Alternatively, the user can create a figure from scratch in the interface, similar to editing a standard binary image. The program reports the total number of pixels covered by each figure to facilitate counting.

**Physical implementations.** There are several ways to manufacture the dissection puzzles we created. If a square lattice is used, we can build the resulting pieces using Lego bricks, which are easy to construct. For triangular lattice or 3D dissections, we produce the puzzles using 3D printing. Figure 2.1 shows several examples of physical puzzles we generated.
Figure 2.10: Three examples of two-figure dissections using the triangular lattice. The examples in (b) and (c) dissect an English letter with an object figure whose name starts with that letter.

2.5 Results and Discussions

Optimality. To examine the optimality of our algorithm, we compared our solutions for several representative dissection problems with the reference solutions described in Frederickson’s book entitled *Dissections: Plane and Fancy* [44]. These examples are demonstrated in Figure 2.12. As shown in the left column, all input figures are rectilinear polygons of integer coordinates, which can be exactly represented using a square lattice. Thus they provide a direct evaluation of our method. The optimal solutions for these examples are known and are listed in the right column of the figure. The middle column shows our solutions. Several inputs contain disconnected components, for which our algorithm can handle successfully. For all examples we achieve the same number of pieces with the reference. Note also that for many of them our solution is different from the reference in terms of the shapes of the resulting pieces. The examples in Figure 1.3 and 2.5 were also produced using our algorithm, and the results are known to be the optimal.

Performance. Our results were obtained on an Intel Core i7 2.66 GHz CPU with 8 hyperthreads. For relatively simple shapes, such as the 2D examples in Figure 2.1 and Figure 2.12, the total computation time is within 20 minutes. The figures in these examples generally contain 50~160 pixels. Higher resolution input will result in increased computation time, but we have found that the cost is more dependent upon the number of levels (hence the number of pieces) required to solve a dissection and less dependent on the number of pixels. This is mainly because each higher level needs to process more clusters. In
addition, multi-figure dissections generally take much longer time to run. For example, our longest computation time is 5 hours for the three-figure Chinese character dissection in Figure 2.9, which produced 13 pieces. All other 2D examples were computed within half an hour. For 3D examples, the one in Figure 2.7 takes around 4 minutes and the one in Figure 2.8 takes around 70 minutes.

Two-figure dissections. Figures 1.3, 2.5, 2.6, 2.12 all demonstrate examples of two-figure dissections using a square lattice. The top two examples in Figure 2.1 were also computed as two-figure dissections, but we have found other interesting figures that each solution set can construct and included an example for each. The bottom example in Figure 2.1 shows a two-figure dissection computed using a triangular lattice, and Figure 2.10 show several additional examples.

Area-based term. In Figure 2.6 we have shown that enabling the area-based term often leads to results where the size of each piece is more balanced. In Figure 2.11 we shown
an additional example where our algorithm found multiple solutions at the same level. The best solution is typically selected as the one that gives rise to the smallest area variance of the pieces. But other criteria can also be used to define the best solution.

**Three-figure dissections.** Figure 2.9 shows two examples of three-figure dissections. In the top example, our algorithm found a 6-piece dissection solution for a $12 \times 12$ square, a $16 \times 12$ rectangle with a cross hole in the center, and a $16 \times 9$ solid rectangle. The bottom example is a three-figure dissection of a square, a Chinese character meaning ‘person’, and a simple figure of a person. Our algorithm found a 13-piece dissection of this example. Many Chinese or Japanese characters are hieroglyphic, thus they are suitable for creating dissection puzzles as the character look similar to the figure it represents.

**3D dissections.** Figures 2.1, 2.7 and 2.8 demonstrate 3D puzzles created using our algorithm. In particular, Figures 2.1 third example is inspired by the 2D Pythagorean triples and is a demonstration of $3^3 + 4^3 + 5^3 = 6^3$; and the fourth example is the dissection of a polycube Bunny model and a $6 \times 6 \times 7$ cuboid. We have found these puzzles to be quite enjoyable and challenging to play with. Some of them look deceptively simple, but can take a considerable amount of time to solve.
Figure 2.12: A comparison of our solutions with reference solutions shown in [44]. Some of these examples are visualizations of the Pythagorean triple numbers. The left column shows the input, the middle shows our solution, and the right shows the reference solution. For all examples we achieve the equal number of pieces with the reference.
CHAPTER 3

BOXELIZATION: FOLDING 3D OBJECTS INTO BOXES

Figure 3.1: Folding a car into a cube. Our system finds a collision-free folding sequence.

In Chapter 2, I discussed an algorithm to create dissection puzzles, which can then be used in content creation for 3D printing purpose. In this chapter, I am going to discuss the boxelization method for customized puzzle creation, which folds arbitrary input 3D mesh into box-like shapes. Our method cuts the input mesh into folding units, and connects all units by hinges. The output can be physically folded into a box. Our problem is very similar to the hinged dissection problem, which is a dissection problem with the additional constraint that all dissected pieces are connected by hinges. This problem is more difficult than the classic dissection problems in the sense that it imposes the hinge constraint. In terms of fabrication, it also brings new challenges since the geometry of the hinges must be considered when searching for the hinged dissection.

Since hinged dissection is a very difficult problem, to make it tractable, I relax the problem in the following two ways. Firstly I only consider cases where a user-defined object is dissected into a box-like shape, which makes it easy to describe and evaluate the
folded state. Secondly I do not perform strict dissections, that is, small modifications on the shapes are allowed. Our method is able to solve the relaxed problem, and can produce a single, connected object that can be physically fabricated and folded from one shape to the other, as is shown in Figure 3.1.

The following of this chapter is organized as followed. In Section 3.1, I will talk about related works. In Section 3.2, I describe our algorithm. On a high level, our algorithm handles such problem by segmenting the input shape into voxels and search for a voxel-tree that can fold from the input shape to the target shape. In Section 3.2.1, we discuss the voxelization step, where we discretize the shape into voxels. After voxelization we begin searching for a tree structure which defines the folding, which is described in Section 3.2.2. Since our goal is to design printable shapes, we also discuss how to construct the physical joints. The tree search gives collision-free dissection in the initial and folded state, but there is no guarantee that it is also collision-free during the folding sequence. Finding a collision-free folding sequence for arbitrary shape, however, is a PSPACE complete problem. To handle such problem, we introduce interactive folding in Section 3.2.3, where user interaction is involved. Results of our algorithm is demonstrated in Section 3.3. As is shown, we can fabricate the results using a 3D printer, which gives physically foldable puzzles. The fabricated puzzles are difficult to solve.

3.1 Related Work

Our problem can be divided into several sub-problems such as segmentation (or voxelization), joint placement, and folding. Although there is a rich literature for each of those sub-problems, we are not aware of any works that combines them and solve them together as ours.

One sub-problem is shape segmentation and voxelization, where we cut the object into folding units. Shape segmentation is an active area of research where a surface or volumetric mesh is cut into pieces based on geometry or semantic attributes [108, 32]. And
voxelization discretizes 3D objects into small units, which is useful for physical simulation and analysis, for medical imaging and visualization, and for computer graphics and games [123, 100, 76, 29]. In our setting, our voxelization is used for the fabrication and geometric-fitting purposes, which has not been discussed by existing techniques.

Another related problem is folding. Foldable designs have long been created for furniture and other useful objects (umbrellas, chairs, tents etc.). Our domain is closer to recreational puzzles and art forms such as popup books [72, 71], papercraft toys [87], and cubebots [129]. Folding of paper to create various shapes (Origami) has been studied extensively [96]. Folding has also been extended to developable surfaces with curved folding [60], and to the creation of polyhedral surfaces [116]. Our work can be seen as a type of voxel-Origami (or “ori-voxel”) since, once we find a solution, we can begin from simple boxes and fold them into various 3D-shapes.

Finding a folding pattern can become a very challenging problem [13] and in some cases even present an intractable search space. This complexity also appears in related fields such as protein folding [22, 58]. Demaine and O’Rourke [40] summarized some very nice mathematical results for folding linkages, planes, and polyhedra. Our specific problem is close in spirit to linkages, but in our case, the parts, configuration, and structure of links are unknown as well.

Our work is also in the domain of puzzle designing. In computer graphics community, recently many works have been proposed to create different types of puzzles. These include polyominoes [75], burr puzzles [133], interlocking puzzles [111], dissection puzzles [139], or sliding planar slices [53]. All these works create disjoint-pieces puzzle, which is different from our problem where we seek a single connected object folding into two shapes. Also, in our problem, the addition of joint constraint to keep the pieces connected presents new challenges not encountered in previous methods.

At last, our work is proposed for the purpose of 3D printing and fabrication. Computer assisted fabrication of objects is a new area of research emerging from graphics, CAD,
3.2 Methodology

Our method contains the following steps. Firstly, to segment the shape into small folding units, we run the voxelization step. The voxels can be represented as a connectivity graph. Then, to find the folding structure, we search for a joint tree within such graph. An
overview of those steps is shown in Figure 3.2. The last step, interactive folding, is applied as a post-process to check if the design is physically plausible and can be fabricated.

3.2.1 Voxelization

The first step in our approach is to find a voxelization of the input shape that will meet our objectives. Voxelization is performed by placing a grid around the object and marking the voxels that contain any part of the object. By intersecting the voxels with the object mesh we create the set of pieces for folding. For convenience, we continue to call these pieces “voxels”, even though some of them are only partially filled voxels.

We use cube-shaped voxels as they allow full freedom of movement in folding and placing of hinges. Hence, the free parameters for voxelization are the dimensions of the grid and its position and orientation in space. Because our target shape is a box or a template that can fold into a box, we fit the dimensions of the grid so that the number of pieces will be equal or smaller than those of the box (we used several box sizes from $3 \times 4 \times 4$ to $5 \times 5 \times 5$). We therefore search only for the orientation and position of the grid. In addition, we allow small deformations of the input object to optimize the fit into the voxels as will be described below. In general, the voxelization grid can also be defined and positioned manually by the user.

To meet the printability criterion, our main goal in voxelization is to make sure that the actual volume of each final voxel piece is large enough to support and hold the connecting hinges and be printable. Moreover, the closer the shapes of the pieces are to full voxels, the easier it would be to fill a target box shape with little waste of space. Hence, we define the “fullness” objective function as follows:

$$E_{\text{vox}} = \sum_{v \in V} \left\{ \begin{array}{ll} 0 & \text{if } M \cap v = \emptyset, \\ 1 - \frac{\text{volume}(M\cap v)}{\text{volume}(v)} & \text{otherwise}, \end{array} \right.$$  (3.1)
where $M$ is the input mesh, $V$ is the voxelization, and $v$ is a voxel. $M \cap v$ is the intersection of $M$ and $v$. Although mesh intersection can be used to compute the volume, we instead use a voxelization approach once again. After the grid is chosen, we subdivide the grid further so that each voxel is composed of 20x20x20 subvoxels (3.3(d)). The volume of intersection between the mesh and a voxel, $M \cap v$, can then be efficiently approximated by counting the subvoxels occupied by the mesh inside each voxel. These subvoxels also used for non-uniform voxelization and the evaluation of the folding objective function, described below.

The graph of Equation 3.1 is shown in the inset figure. This function penalizes voxels occupied by a small portion of the input mesh but does not penalize empty voxels. We do not need a threshold since the volume computation using subvoxels means the volume ratio takes on discrete values. To optimize this function we choose different randomized rotation and translation of the grid, and keep the best results after applying the non-uniform voxelization step.
**Non-uniform voxelization.** To lower the objective function further we allow slight deformation to the input shape by locally offsetting the grid planes, as shown in 3.3(b-c). We limit the offset of the grid planes so that the grid spacing does not change by more than 10-20% along the normal of the plane, to ensure that the distortion to the input shape would be small. This corresponds to each of the planes having the freedom to move $\pm 2 - 4$ sub-voxels. The X, Y, and Z planes are adjusted using a block coordinate descent approach; we hold two of the directions fixed and adjust the planes in the remaining direction. Optimal solution in terms of the energy function (3.1) is found using dynamic programming in each direction. We iterate between the three directions (X, Y, Z, X, Y, Z, ...) until convergence, which in our examples tend to be around 3-4 iterations per direction. Once the optimal grid plane offsets are found, they are moved back to their original positions, carrying along with them the input mesh, which results in a slightly deformed mesh with fewer small voxel pieces.

### 3.2.2 Tree Fitting

After the object is segmented into voxels, we need to find the connectivity between the voxels so that the resulting object can be folded into the target shape. The voxels created in the previous step do not yet have any joints between them. However, they do provide the geometric neighborhood information that defines the potential joint locations—we can only add joints between voxels that contain part of the object along their shared face (Figure 3.2b). Our final goal is to define an undirected tree to represent the connectivity between the voxels: nodes correspond to the voxels, and edges correspond to joints that connect the voxels (Figure 3.2c). As mentioned earlier, we do not allow loops, as they almost always create over-constrained configurations. The objective of the fitting step is to find a low energy tree that spans all the voxels by assigning a joint type to each pair of neighboring voxels. The energy we use will be defined later in this section. The joints are parameterized by the following types:
• **Null**: No joint is added between the voxels and they can be separated. These correspond to the dotted edges in Figure 3.2b that were removed in Figure 3.2c.

• **Rigid**: The nodes, and the voxels that they represent, are attached rigidly. This means there is no hinge between these voxels and they move together. These correspond to the thick edges in Figure 3.2c.

• **Single hinge**: A simple hinge that connects the voxels with a single axis of rotation, as shown in Figure 3.5a (top left). There are 4 types of single hinges, corresponding to the 4 rotation directions of the child voxel with respect to the parent voxel.

• **Double hinge**: A hinge with two axes of rotation connecting the voxels. This joint type provides a rich set of transformations of the child with respect to the parent, some of which are shown in Figure 3.5.

Using this parameterization, the search boils down to assigning a joint type to each graph edge in Figure 3.2b so that the end result is a tree, as in Figure 3.2c.

The double hinge provides a rich set of transforms for the tree fitting stage while still being simple enough for physical printing. We parameterize the double hinge by the two axes of rotation it provides: the 1st axis between the parent voxel (shown in pink in Figure 3.5a) and the link body (green), and the 2nd axis between the link body and the child voxel (purple). The parameterization can be described compactly as “[axis][sgn]:[axis][sgn]”, where [axis] can be X, Y, or Z, and [sgn] can be -, - -, +, or ++. We use “-” to indicate a -90° rotation, “- -” for -180°, “+” for +90°, and “++” for +180°. For example, the 3 double hinges in the figure are Z-:Z-, Z-:Y-, and X+:Z-. A sample transform is shown in Figure 3.5b—with respect to the parent voxel, the child voxel translates to the +Z position and rotates by -90° around the Z-axis. With a double hinge, a child voxel can be transformed to a total of 78 distinct axis-aligned configurations in \( SE(3) \), after all the double counting
Figure 3.5: (a) Examples of hinge types. The X-axis is to the right, Y is into the paper, and Z is up. Top row: “Y-”, “Y+:Y++” & “Z++:Y-”. Middle row: “Z-:Z-”, “Z-:Y+” & “Z-::Y-”. Bottom row: “Y-:Z-”, “Y++:Z-” & “X+:Z-”. (b) Example motion sequence of a double hinge.

has been accounted for (e.g., Y++:Y++ and Y-::Y- give the same transform). This is in contrast to the single hinge, which only provides 4.

We can now define the search space formally. Let \( x_i \) be the joint type of the \( i^{th} \) edge. Then the assignment of edge types can be expressed as

\[
x_i \in \{N, R, S_{Z+}, \ldots, D_{Z+:Z+}, \ldots\}, \quad i = 1, \ldots, n,
\]

where \( n \) is the number of edges, and \( N, R, S, \) and \( D \) correspond to the joint types listed above.

Let \( V_0 \) be the transform of the root node of the tree, which is chosen randomly. Given a sequence of joint types, \([x_1, x_2, \ldots]\), starting from the root transform, we can compute the transformation of each voxel, \( V_i \), by traversing the tree from the root to the voxel.

---

\(^1\)The total number of axis-aligned configurations is 144. There are 6 different positions for the child with respect to the parent: \( \pm X, \pm Y, \pm Z \). For each of these positions, there are 6 different ways in which the X-axis of the child can point, and after that 4 more choices for the Y-axis.
\[ V_i = \text{VoxelTransform}(V_0, [x_1, x_2, \ldots]). \quad (3.3) \]

We use \( V_i \) to denote the transformation of the \( i^{th} \) voxel, i.e., the 4x4 \( SE(3) \) matrix that transforms from voxel’s local coordinates to world coordinates. Depending on the context, we also use \( V_i \) to denote the final position of the \( i^{th} \) voxel in \( \mathbb{R}^3 \). We also use a similar traversing function to compute the position and orientation of the \( i^{th} \) joint.

\[ J_i = \text{JointTransform}(V_0, [x_1, x_2, \ldots]). \quad (3.4) \]

If we randomly assign values to the edges, then the resulting folded configuration will almost always suffer from collisions. Instead, we build a collision free configuration incrementally using a tree search. Starting from a randomly chosen root node, the fitting step advances on the graph using beam-search, an extension of best-first search that sorts and keeps the top partial solutions whenever a new search path is explored. Unlike breadth-first search and its variants, beam search keeps the memory footprint small by throwing away paths that look to be the least promising. The tree is expanded one edge at a time while keeping the resulting partial configuration collision free. The search ends when the tree spans the voxels and all edge types have been determined.

**Fitting Energy.** The energy is a function of the root transform and the sequence of edge types: \( E(V_0, [x_1, x_2, \ldots]) \). As we build the tree, we evaluate the energy whenever the tree is expanded by adding an edge. Initially, the tree only contains the root node, so the energy is \( E(V_0, []) \), and only the transform of the root is known. Then, the edges incident to the root, which is the current frontier, are evaluated, and the most promising ones are added to the frontier. For brevity, we use \( E(\mathbf{x}) \) to indicate \( E(V_0, [x_1, x_2, \ldots]) \).

The energy function has four terms.

\[ E = E_{\text{collision}} + E_{\text{template}} + E_{\text{surface}} + E_{\text{count}}. \quad (3.5) \]
The first two terms are hard constraints, and the last two are energy objectives.

**Collision.** The collision term constrains the folded shape from placing voxels or joints at the same location in space: $E_{\text{collision}} = E^V_{\text{collision}} + E^J_{\text{collision}}$. We do, however, allow for two partially-filled voxels to be at the same location if their meshes do not overlap when placed at the same location. The joint collision term is required to prevent two single hinges to reside on the same edge of the voxel, or from two double joints to originate from the same voxel.

$$
E^V_{\text{collision}}(x) = \begin{cases} 
\infty & \text{if } V_i(x) = V_j(x), \\
0 & \text{otherwise}, 
\end{cases}
$$  

$$
E^J_{\text{collision}}(x) = \begin{cases} 
\infty & \text{if } J_i(x) = J_j(x), \\
0 & \text{otherwise}, 
\end{cases}
$$  

(3.6)

for some $i$ and $j$. The equality in this equation only checks for the positions of voxels $V_i$ and $V_j$ and not their orientations. Voxel collisions are trivial to compute using the subvoxels computed in the voxelization step from Section 3.2.1.

**Template.** The template term constrains the folded shape to match the target template (the box) and is again composed of two subterms that correspond to voxels and joints: $E_{\text{template}} = E^V_{\text{template}} + E^J_{\text{template}}$. A template defines sets of positions, $T_V$ and $T_J$, that the folded voxels and joints, respectively, are allowed to take.

$$
E^V_{\text{template}}(x) = \begin{cases} 
\infty & \text{if } V_i(x) \notin T_V, \\
0 & \text{otherwise}, 
\end{cases}
$$

$$
E^J_{\text{template}}(x) = \begin{cases} 
\infty & \text{if } J_i(x) \notin T_J, \\
0 & \text{otherwise}, 
\end{cases}
$$

(3.7)
Constraints on the placement of joints imposed by partially filled voxels are included in the joint template term. For instance, a single hinge cannot be constructed on a voxel piece unless the edge it is assigned to contains a large enough part of the object so as to position the hinge geometry on. Since most voxels are only partially filled, this constrains the search considerably.

Note that the template and collision energy terms are hard constraints. If they are violated, then the tree search prunes off the branch and searches down another branch. The following two terms are used as soft constraints to differentiate between feasible solutions.

**Surface.** Because our goal is to create a box whose faces should be as planar as possible, we want the outside faces of boundary voxels in the target configuration to be filled. We use the surface energy to encourage this behavior. A 2D illustration is given in the inset figure. The red voxel edges form the boundary surface of the template. Rays, shown in green, are shot from the boundary until they hit the surface or the edge of the voxel. The ray distances are integrated to give the energy for that voxel. The energy is minimized when the shape matches the boundary and is maximized when the voxel location is unoccupied.

\[
E_{\text{surface}}(x) = \sum_i \int \text{ray distance}.
\]  

(3.8)

In the inset figure, the top two voxels have high energy, the lower right voxel has low energy, and the lower left voxel has zero energy. For interior voxels that do not contain a border, we set the energy to be zero. Instead of actually shooting rays and calculating distances we use the subvoxels from Section 3.2.1.
**Counting.** The final energy term counts the number of joints. Whenever possible, we prefer solutions with a fewer number of joints as it will make folding simpler. Furthermore, some joint types are preferred over others since they require less modification to the input mesh. Each joint type is given a weight, and we simply sum the weights to compute the energy.

$$E_{\text{count}}(x) = \sum_i \|x_i\|,$$

(3.9)

where $\| \cdot \|$ denotes the numerical weight given to each joint type listed in 3.2.

**Simulated Annealing.** The tree fitting step returns a list of solutions ordered by the energy value. Since the first two energy terms are hard constraints, these solutions are guaranteed to be collision free and to fit inside the template. Usually, however, just one tree search does not give a satisfactory solution—some solutions have poor surface energy, and others have too many joints to be printable. Therefore, we combine the tree search with simulated annealing. Initially, the annealing temperature is set to be high, which means that the tree search is run many times with random position and orientation of the root voxel. This portion of the algorithm is embarrassingly parallelizable. After we have a certain number of solutions, we lower the temperature gradually, so that whenever a good solution is found, we start the search using a partial subtree from that solution.

**Geometric Post-processing.** Once the joint types are determined, we must modify the voxels to include the geometry of the joints. We are guaranteed not to have any hinges on an empty edge of a voxel, because of the hard constraints applied in the tree search. We must also carve out some geometry from the voxels to enable proper motion of the joints. As shown in Figure 3.5a, a single hinge is less obtrusive than a double hinge to the voxel geometry, requiring less of the voxel to be carved out. At this stage, we only look at neighboring voxels. Sometimes, it is necessary to carve out the corners of the voxels due to global contacts, and this is addressed in the next section.
3.2.3 Interactive Folding

The tree search only considers collisions in the folded state and not during the movement of the voxels in the folding sequence. This means that the computed solution may not be physically foldable when manufactured. We mitigate this problem by disallowing loops in the connectivity graph, but we must still verify that the computed solution can be folded without collisions. We use a semi-automatic approach that combines a physical simulator and user interactions. The key idea here is that physics is quite effective at unfolding even though it does not work well for folding.

The process of folding the original shape (Shape A) into the target shape (Shape B, for “B”ox) is broken up into two steps as shown in Figure 3.7: unfolding and matching. First, the simulator simultaneously tries to unfold both the original shape (A) and the folded shape (B) by applying a repulsive force ($\propto 1/r^2$) between all pairs of voxels within the shapes. The simulator can be any off-the-shelf rigid body dynamics engine that supports
joint constraints and collisions. At any time, the user can guide the system by supplying additional external forces or by pinning certain voxels strategically.

After both shapes have been unfolded adequately, the repulsive forces are removed and attractive forces between the corresponding voxels from A and B are added to match their shapes. Once A and B take on the same configuration, we have a valid folding sequence from A to B, passing through the intermediate unfolded configuration. Note that the order of first unfolding and later matching is very important. Without unfolding first, the matching force will almost always cause the shapes to get stuck due to collisions and will not be able to cause A and B to reach the same intermediate configuration.

**Interaction.** Figure 3.8 shows a stage in the unfolding process where the physics simulator has managed to unfold most of the joints but is not able to untangle a small portion of the shape. The user intervenes and decides that the correct ordering to unfold is to rotate the purple voxel about the axis labeled “1.” However, before the purple voxel can be rotated, its corners must be carved out, since otherwise collisions will constrain the rotation physically (Figure 3.8b). After carving the appropriate corners, the physics simulator can continue to unfold the remaining voxels (Figure 3.8c). It took around 5-10 minutes of interaction to obtain a physically foldable solution for all results in this paper.

The interactive physics simulator acts as a filter that semi-automatically removes physically unfoldable solutions. If we find a valid folding sequence using the interactive simulation process, then we know that the solution is valid. Note however, that if we cannot find it, we cannot guarantee that there is no solution. Also, if there is a valid solution, we are not guaranteed to find it. In practice, this simulator did assisted in filtering out some implausible solutions found in the tree search.

### 3.3 Results

We used our system to create a foldable bunny (Figure 3.9a), kitten (Figure 3.9b), car (Figure 3.9c 3.10a), dragon (Figure 3.10b), and elephant (Figure 3.10c). For all our ex-
Figure 3.8: (a) The user sees that the correct unfolding sequence is to rotate the purple voxel around 1 and then the pink voxel (which is the purple voxel’s parent) around 2. (b) But the corner (marked in yellow) on the purple voxel prevents rotation. (c) The user carves some edges of the purple voxel and continues unfolding.

amples, we used the following energy weights: \( w_{\text{surface}} = 0.3 \), \( w_{\text{count(N)}} = 0 \), \( w_{\text{count(R)}} = 0 \), \( w_{\text{count(S)}} = 0.1 \), and \( w_{\text{count(D)}} = 0.12 \). The first three results were physically manufactured using the Objet500 Connex 3D printer. For the tree search, we run the search in parallel using a cloud computing service. We ran the tree search for up to \( \sim 30 \) hours with random restarts for each object. Then we sort the generated solutions by the energy (all hard constraints are satisfied), and starting from the best solution, we interactively folded for around 5-10 minutes until we found a good working solution.

Table 3.1 lists the running time of the tree search, the number of joints in the computed solution, and the number of voxels carved during the interactive folding session. “Wall” is the wall-clock time to find the solution used in the examples, “CUs” is the number of normalized compute units employed (roughly equivalent to a single 1GHz core), and “total” is the product of these two numbers, which is the total number of core hours. Note that these numbers represent the amount of time the tree search took to find the solutions used for the results, not the total time we ran the tree search (up to \( \sim 30 \) hours). As can be seen, a reasonable solution can be found in the 5-15 hour range, depending on the example. As this process is embarrassingly parallelizable, these numbers can be further reduced by using more processing power.
We found the animation produced by the interactive simulation, as shown in Figure 3.1, to be extremely useful as a guide for folding and unfolding the 3D-printed prototypes (car, bunny, and kitten). Even for the simplest example, it is non-trivial to fold the shape from one to the other without the aid of the provided animation.

The U-car example shown in Figure 3.10a demonstrates that shapes other than cubes can be used as the target template, as long as it is a voxelized shape.

For the dragon shown in Figure 3.10b, the best results in terms of energy were generated for a voxelization that has some voxels that cover disconnected pieces. In these cases, we added “struts” between disjoint pieces.

The final example, the elephant shown in Figure 3.10c, uses a $5 \times 5 \times 5$ voxelization.

Table 3.1: “Size” is the voxelization resolution. “wall” is the number of wall-clock hours to find the solution used in the examples. “CUs” is the number of compute units used (roughly equivalent to a single 1GHz core). “Total” is the total number of compute hours, i.e. the product of wall and CUs. “Joints” is the number of joints that are added to the shape. “Carves” is the number of voxels that are carved with the interactive simulator.

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>wall (h)</th>
<th>CUs</th>
<th>total (h)</th>
<th>joints</th>
<th>carves</th>
</tr>
</thead>
<tbody>
<tr>
<td>bunny</td>
<td>4×4×4</td>
<td>4.9</td>
<td>50</td>
<td>245</td>
<td>29</td>
<td>7</td>
</tr>
<tr>
<td>kitten</td>
<td>4×4×4</td>
<td>6.9</td>
<td>50</td>
<td>345</td>
<td>34</td>
<td>3</td>
</tr>
<tr>
<td>car</td>
<td>3×4×4</td>
<td>11.5</td>
<td>14</td>
<td>161</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>U-car</td>
<td>112</td>
<td>16.1</td>
<td>50</td>
<td>805</td>
<td>45</td>
<td>19</td>
</tr>
<tr>
<td>dragon</td>
<td>4×4×4</td>
<td>4.3</td>
<td>50</td>
<td>215</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>elephant</td>
<td>5×5×5</td>
<td>13.4</td>
<td>75</td>
<td>1005</td>
<td>62</td>
<td>7</td>
</tr>
</tbody>
</table>
Figure 3.9: Results physically manufactured using a 3D printer.
Figure 3.10: (d) The U-car demonstrates that the target does not need to be a box. (e) The dragon contains some struts due to the challenging geometry. (f) The $5 \times 5 \times 5$ elephant is the largest example produced.
CHAPTER 4
DIRECT SHAPE OPTIMIZATION FOR STRENGTHENING PRINTABLE OBJECTS

Figure 4.1: A stool before and after optimization. The middle part is deformed in order to stand more force.

In Chapter 2 and 3 I discussed content creation for 3D printing by automatic puzzle design. In this chapter I am going to discuss another aspect for content creation, that is to optimize existing designs for their functionality. Existing modeling software is often designed for experienced users with extensive training, and lacks the capability to guide novice users towards designing functional and physically plausible shapes. Consider a chair model, when printed, it may not sufficiently withstand forces during its everyday use; a small RC jet may have poor aerodynamic properties.

In this chapter, I will describe a new 3D modeling algorithms that can assist users to quickly design functional shapes. The algorithm will use an optimization-based framework
— it performs geometric shape deformation to iteratively reduce mechanical stresses in the presence of forces during typical use scenarios, as is shown in Fig. 4.1.

The following of this chapter will be organized as followed. Firstly I will briefly go over the background knowledge about physics simulation and transformed surface Laplacian in Section 4.1. Related works are discussed in Section 4.2. In Section 4.3, I formulate our problem mathematically. Our solution is discussed in Section 4.4, which is composed of the following parts: simulation constraints (Section 4.4.1), stress constraints (Section 4.4.2), geometry constraints (Section 4.4.3), objective function (Section 4.4.4) and optimization solver (Section 4.4.5). A variety of results are demonstrated in Section 4.5, where I show our optimization results for tools and furnitures. To validate my algorithm, I also print some of my results and physically test their strength. The new modeling algorithm can significantly shorten the design cycle of 3D products, enable rapid prototyping, and make it easy for users to create and print complex functional shapes.

The above algorithm assumes that the input mesh can be nicely tetrahedralized or hexahedralized. However, this assumption does not apply to general shapes designed by inexperienced users. The quality of tetrahedralization/hexahedralization will highly affect the optimization result. Thus, in Section 4.6, I introduce the SPH-bases sampling method, where it can produce blue noise samples within a volume as well as on the surface of the design. Experimental results show that such technique provides high quality results with uniformly distributed elements in the remeshing task. With such technique, we can tetrahedralize designs uniformly with smaller number of elements, which improves the accuracy of physics simulation and consequently results in an improvement in the quality of the optimized design.
4.1 Background

In this section we will briefly go over the background knowledge of physics simulation and transformed surface Laplacian, which are required to understand the content of our solver.

4.1.1 Physics Simulation

In this section I will give a brief overview of physics simulation in computer graphics, which will serve as the foundation of the technology used later in this chapter. Readers who are interested in more details can find a throughout discussion about physics simulation in the Real-time Physics course note [93].

Physically based simulation has been developing for decades in computer graphics for the purpose of creating physically plausible animations. Many techniques are develop to simulate the behavior of different types of objects, such as solid, fluid, smoke, flame, etc. In this chapter, since we aim at optimizing a design, we will focus on solid object simulation. More specifically, our simulation falls into the category of soft body simulation. Soft body simulation predicts the deformation of an object under external forces and constraints. In compute graphics, such technique is usually applied to give realistic animation of elastic or plastic deformation for movies and games. It can also be used to predict whether the object can serve its designing purpose properly for the purpose of design and fabrication. In this chapter, we will use it for such purpose.

Among all the soft body simulation methods, finite element method (FEM) is the mainstream method. The benefit of this method is that it gives high simulation accuracy especially when simulating elastic deformations. The drawback of such method is when the object undergoes high plastic deformation and with high stress, remeshing is usually required, which is time-consuming. Other soft body simulation methods include point-based dynamics, which can easily handle topology changes but cannot offer competitive accuracy.
with FEM. Since our design optimization always try to stay with elastic deformation and avoid plastic deformation, we will focus on FEM.

**Finite Element Method.** In this section I will give a brief overview of the finite element method. The purpose of this method is to describe the deformation of an object and calculate the elastic force that is caused by such deformation. With the elastic force, we can simulate its deformation in the next timestep (for dynamic simulation), or calculate its deformation in the force equilibrium state (semi-static configuration).

The method begins by discretizing the object into a collection of basic elements, i. e., tetrahedra or hexahedra. The benefit of such representation is that we can approximate the deformation gradient as a linear or constant function within each element, which largely simplifies the problem by reducing the partial differential equations for deformation into a system of algebraic equations. The discretization step will usually be treated as pre-process and can be solved with existing softwares [120] [109]. In the following we only discuss the case where the object is discretize into tetrahedral elements.

Figure 4.2: An illustration of a deformed tetrahedron. The vertex \( x_0 \) is deformed to \( p_0 \).
Once we discretize the whole model into basic tetrahedra, we can focus on the deformation and elastic force within each tetrahedron. To describe the deformation, one needs to deal with two coordinates for the object: the material coordinate $X$, which describes the rest state of the object without any external forces; and the world coordinate $P$, where the simulation actually happens and the animation of the object at each timestep is described. For a tetrahedron $t$ with vertices $v_0, v_1, v_2, v_3$, by calculating the difference between its two coordinates, one can figure out how much it deforms. Specifically, let us denote the material coordinate of the 4 vertices as $x_{v_0}, x_{v_1}, x_{v_2}, x_{v_3}$, and their world coordinate $p_{v_0}, p_{v_1}, p_{v_2}, p_{v_3}$, as is shown in Figure 4.2. One can build 3x3 matrices $X = [x_{v_0} - x_{v_3}, x_{v_1} - x_{v_3}, x_{v_2} - x_{v_3}]$ and $P = [p_{v_0} - p_{v_3}, p_{v_1} - p_{v_3}, p_{v_2} - p_{v_3}]$. Then we can easily compute a linear mapping $F$ from the material coordinate to the world coordinate:

$$F_t = P_t X_t^{-1}. \quad (4.1)$$

We call $F_t$ the deformation gradient. Here $t$ denotes the tetrahedron. With a proper offset, this mapping can map all position in the material coordinate to its corresponding world coordinate. This can be easily verified because for all $p_{vi}$ we have:

$$p_{vi} = F_t(x_{vi} - x_{v3}) + p_{v3} \quad (4.2)$$

Whenever a material undergoes elastic deformation, according to Hook’s law there would be an elastic force trying to restore it to its original shape, which is what we need to compute for our simulation. To compute it, we describe the relative elongation or compression of the material as the strain tensor $\epsilon_t$, which is a function of deformation gradient. Traditionally people consider two ways to compute the strain tensor:

$$\epsilon_t^G = \frac{1}{2}(F_t^T F_t - I), \quad (4.3)$$

$$\epsilon_t^C = \frac{1}{2}(F_t^T + F_t) - I. \quad (4.4)$$
The first one is called the Green strain tensor, and the second one the Cauchy linear strain tensor. Notice that both tensors give zero value when there is no deformation, which conforms with our intuition.

To evaluate the force, we can begin with the definition of elastic strain energy:

\[ U_t = \frac{\mathcal{V}_t}{2} \epsilon_t \otimes E \epsilon_t. \]  

(4.5)

where \( \mathcal{V}_t \) is the volume of the tetrahedron \( t \) in the material coordinate. \( E \) is a tensor determined by the property of the material. Here \( \otimes \) denotes tensor multiplication. We also define the stress tensor \( \sigma_t \). For the \((i,j)\) component in the stress tensor \( \sigma_{ij} \), we can define the tensor multiplication:

\[ \sigma_{ij} = \sum_{k,l} E_{ijkl} \epsilon_{kl}. \]  

(4.6)

With the elastic strain energy, the elastic force over each vertex on the tetrahedron can be computed as the derivative of energy over vertex position:

\[ f_{t,v} = \frac{\partial U_t}{\partial p_i} = \mathcal{V}_t \sigma_t \otimes \frac{\partial \epsilon_t}{\partial p_v}. \]  

(4.7)

Notice that with the Cauchy linear strain tensor, the force \( f_v \) is a linear function of world coordinate \( P \).

Till now we describe the elastic force from one tetrahedron. Usually a vertex is on multiple tetrahedra, in this case we need to sum up the elastic force from all those tetrahedra:

\[ f_v = \sum_{\forall t, v \in t} f_{t,v} + f_{v}^{ext}. \]  

(4.8)

Here \( f_{v}^{ext} \) refers to external force load. With the pre-vertex force we can predict the animation of the object under Newton’s laws.
Semi-Static Configuration. Previously we discussed how to compute the deformation and force over a tetrahedral mesh. To analyze the force load and stress distribution over a design during usage, we usually apply the semi-static assumption. Semi-static analysis assumes that during usage, every part of the design will reach force equilibrium at each timestep. Such assumption holds if the design does not have high acceleration during usage. For our purpose of strengthening a design, this assumption generally holds in our use cases and will be applied in our work.

With this assumption, we will have the force equilibrium when calculating vertex forces:

$$\forall v, f_v = 0.$$  \(4.9\)

This condition, together with external force load and fix boundary condition, gives us a system that can be solved for the world coordinate of each vertex \(P\). With \(P\) we can also easily compute the corresponding strain \(\epsilon\) and stress \(\sigma\).

Von Mises Stress. So far we discussed how to compute force, strain and stress. But we did not discuss how to determine whether a design will fail or not under some force load. This can be approximated by evaluating the von Mises stress throughout the whole design. The von Mises stress is a function of stress:

$$\hat{\sigma} = \sqrt{\frac{1}{2}[(\sigma_{1,1} - \sigma_{2,2})^2 + (\sigma_{2,2} - \sigma_{3,3})^2 + (\sigma_{3,3} - \sigma_{1,1})^2 + 6(\sigma_{2,3}^2 + \sigma_{3,1}^2 + \sigma_{1,2}^2)]} = \sqrt{(\sigma_1 - \sigma_2)^2 + (\sigma_2 - \sigma_3)^2 + (\sigma_3 - \sigma_1)^2},$$

where \(\sigma_{i,j}\) refers to the \((i, j)\) component in the stress tensor. \(\sigma_1, \sigma_2\) and \(\sigma_3\) denote 3 principal stresses. For each elastic material there would be an approximated von Mises criterion, which is determined by the material property. Wherever its von Mises stress exceeds such criterion, it is likely to have plastic deformation or suffer from mechanical
failure. In another word, the design will fail when its von Mises stress exceeds the criterion somewhere. Thus, this can work as a standard for us to evaluate whether the design will fail or not.

In summary, our goal here is to ensure the stress in a design does not exceed the criterion. To do so, we firstly compute its world coordinates under semi-static setting, then for each tetrahedron in the model we compute its von Mises stress and compare it with the criterion. If any of them exceeds the criterion, the design will fail. In our work to strengthen the design, such criterion gives us the hint about how stable the design is, and will be used as a target in our optimization.

4.1.2 Surface Laplacian Distance

In this section we will discuss background knowledge about the transformed Laplacian distance [112], which will be used as the intrinsic distance in Section 4.4.4. This function is designed to preserve geometry details of a surface mesh while allowing for local rotation and scaling, as is shown in Fig. 4.3. The transformed Laplacian distance we discuss here is based the one proposed by [112]. The difference between ours and the original method is that, ours allows for not only local rotation and scaling, but also other kinds of align transformations. Also notice because that this metric is designed for surface mesh, all vertices we discuss in this section are surface vertices.

Consider one could extract the local transformation matrix for a vertex on a deformed mesh. If the deformed mesh remains similar with the origin in terms of local features, then the local Laplacian should also match in both meshes after applying the local transformation. Thus, the difference of local feature can be evaluated as the difference in the transformed Laplacian.

In details, To compute the transformed Laplacian, firstly we extract the Laplacian for each vertex in both the origin and the optimized mesh, which is denoted as $\mathcal{L}(x_i^0)$ and $\mathcal{L}(x_i)$. A local transformation $T_i$ is then computed from its neighbours, which optimally
translates, rotates, and scales the original surface Laplacian of each vertex $i$ before comparing it to its surface Laplacian on the optimized surface. For a shape to remain locally similar after local scaling/rotation, the transformed Laplacian should be related to the original one by $T_i$. Thus, mathematically, we can define the transformed Laplacian dissimilarity functional as:

$$D_{intrinsic}(X, X^0) = \sum_{i \in S} \| T_i \mathcal{L}(x^0_i) - \mathcal{L}(x_i) \|^2.$$  \hspace{1cm} (4.10)

In our application $X^0$ is fixed. Thus this metric can be written as a quadratic function of $X$, which will be shown later. Here $T_i$ is the transformation matrix calculated from the $n$-ring neighbour of vertex $v$ that provides the best match for its $n$-ring neighbours on mesh surface. In [112], $T_i$ is restricted to be a matrix with rotation, translation and uniform scaling only. But in our application when designing tools or functional parts, non-uniform scaling is usually allowed. Thus we relax it to an affine transformation. Figure 4.3 illustrates the possible transformations.

![Figure 4.3: Possible Laplacian transformations. The left image shows the original shape. The middle image shows rotation, and the right image shows non-uniform scaling.](image)

Also notice that by definition our transformation matrix is very similar to the deformation gradient in physics simulation context. The difference is that the transformation matrix is calculated from the $n$-ring neighbour of a vertex, while a deformation gradient is usually computed from a tetrahedral.
The computation of $T_i$ is as followed. Let’s denote the $n$-ring neighbour of vertex $i$ as $N_i$. $T_i$ is an affine transformation matrix:

$$T_i = \begin{bmatrix} t_{i,1} & t_{i,2} & t_{i,3} & t_{i,4} \\ t_{i,5} & t_{i,6} & t_{i,7} & t_{i,8} \\ t_{i,9} & t_{i,10} & t_{i,11} & t_{i,12} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$  \hspace{1cm} (4.11)

$T_i$ denotes the optimal transformation from the original design to the deformed one, which means it minimizes the L-2 error between those two regions under such transformation. There are 12 free parameters in $T_i$ because it is an affine transformation matrix. Thus, $T_i$ can be mathematically written as:

$$T_i = \arg \min_T \left( \sum_{j \in N(i)} \| T x_0^j - x_j \|^2 + \| T x_i^0 - x_i \|^2 \right).$$ \hspace{1cm} (4.12)

Now we discuss how to simplify Equation 4.10 into a quadratic form. We firstly concatenate all vertices into a long vector and rewrite $T_v$ in vector form:

$$\begin{bmatrix} t_{i,1} \\ t_{i,2} \\ \vdots \\ t_{i,12} \end{bmatrix} = \arg \min_{t_{1\ldots12}} \left\| \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{j+k-1} \end{bmatrix} - \begin{bmatrix} x_i \\ x_j \\ \vdots \\ x_{j+k-1} \end{bmatrix} \right\|^2$$ \hspace{1cm} (4.13)

where $x_j \ldots x_{j+k-1}$ are the neighboring vertices, and $k = N(i)$ denotes the number of neighbors. Matrix $C$ is a matrix composed of vertices on the original design. Here we use bold 0 to denote a vector of 3 zeros. Matrix $C_i$ can be written as:

$$C_i = \begin{bmatrix} x_i^0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & x_i^0 & 1 & 0 & 0 \\ 0 & 0 & 0 & x_i^0 & 1 \end{bmatrix}.$$ \hspace{1cm} (4.14)
Such quadratic system can be solved as:

\[
\begin{bmatrix}
    t_{i,1} \\
    t_{i,2} \\
    \vdots \\
    t_{i,12}
\end{bmatrix}
= (C_i^T C_i)^{-1} C_i^T
\begin{bmatrix}
    x_i \\
    x_j \\
    x_{j+1} \\
    \vdots \\
    x_{j+k-1}
\end{bmatrix}.
\] (4.15)

After solving for the transformation matrix, we can substitute it into the transformed Laplacian \( T_i \mathcal{L}(x_0^i) \). Notice that in our cases, \( T_v \) is a linear function of \( X \) and \( \mathcal{L}(x_0^v) \) is a constant. Thus, \( T_v \mathcal{L}(x_0^v) \) also gives a linear function of \( X \), which is in the form:

\[
T_v \mathcal{L}(x_0^i) = D_i \ast \begin{bmatrix}
    t_{i,1} \\
    t_{i,2} \\
    \vdots \\
    t_{i,12}
\end{bmatrix}
= D_i (C_i^T C_i)^{-1} C_i^T
\begin{bmatrix}
    x_i \\
    x_j \\
    x_{j+1} \\
    \vdots \\
    x_{j+k-1}
\end{bmatrix}.
\] (4.16)

Here the matrix \( D_i \) composes of the Laplacian of \( x_0^i \) and its neighbours, which is in the form:

\[
D_i = \begin{bmatrix}
    \mathcal{L}^T(x_0^i) & 0 & 0 & 0 & 0
    \\
    0 & \mathcal{L}^T(x_0^i) & 0 & 0 & 0
    \\
    0 & 0 & \mathcal{L}^T(x_0^i) & 0 & 0
    \\
    0 & 0 & 0 & \mathcal{L}^T(x_0^i) & 0
\end{bmatrix}.
\] (4.17)

Now in Equation. 4.10, both term \( T_v \mathcal{L}(x_0^v) \) and \( \mathcal{L}(x_v) \) are linear functions of \( x_v \) when \( X^0 \) is fixed, thus Equation. 4.31 can be rewritten as a quadratic function:

\[
D_{intrinsic}(X, X^0) = X^T LX
\] (4.18)
Where $L$ is a matrix determined by $X^0$. The quadratic form makes it easy for computation, and the $L$ matrix only needs to be computed once at the beginning of the optimization.

**Ring of Neighbors.** Notice that when calculating Laplacian dissimilarity, each vertex shall have at least 4 neighbors. This is because such function actually allows for free-form deformation if there are only 3 neighbors. For such vertices, $T_i \mathcal{L}(x^0_i) - \mathcal{L}(x_i) \equiv 0$. To avoid such case, in all our experiments we use 2-ring of neighbors.

**Singularity.** Also notice that in Eq. 4.15, one needs to invert $C_i^T C_i$. Such an operation is ill-defined when $C_i^T C_i$ is singular. This corresponds to the case where vertex $i$ and all its neighbours are co-planar. In such case, we would prefer those vertices to remain co-planar after deformation. Thus, in implementation when $C_i^T C_i$ is near singular, we compute its singular value decomposition (SVD), set its close-to-0 singular value to a very small positive number (in our implementation, $10^{-8}$ times the square of average neighboring vertex distance), and compute its inverse matrix. This means to add a huge weight for any deformation along the normal direction of the mesh. Thus it forces a planar vertex to remain co-planar with its neighbours when possible.

### 4.2 Related Work

Our work is related to research on improving the structural strength of 3D printable objects, which aims to diminish the chances that objects will break during common use scenarios. A number of approaches have been proposed towards achieving this goal, including modifications to the interior structure of objects, such as hollowing [114, 79, 122] or internal skin framing [127], addition of supporting struts [114], change of printing directions [54, 121], and shape deformations [114, 121].

The above approaches are complementary to each other. In other words, to reduce stress in objects, a common practice is to use a combination of these approaches. However, there are cases where adding supporting structures or modifying the internal object structure are
undesirable. For example, adding struts can be inappropriate for artistic designs, since they alter the appearance of the designed objects. Hollowing may increase mechanical stress in the presence of large external forces, and sometimes requires users to perform additional post-processing on the printed object e.g., remove internal powder [127]. Changing the printing direction may also not always yield a decrease in mechanical stress [54]. Thus, applying shape deformations remains one of the main options to reduce mechanical stress, and in some cases, the most desirable one. Existing shape deformations for reducing mechanical stress have been so far limited to uniform part thickening. Stava et al. [114] identifies thin, vulnerable, parts in an object and applies a heuristic to increase their thickness assuming that these parts are thin tubes and acting forces cause bending. Vulnerable parts can be alternatively identified with worst-case load analysis methods [137]. Umetani and Schmidt [121] highlight vulnerable areas to users and allow them to interactively thicken these areas. A downside of part thickening is that it increases material use and can affect the appearance of the model in an undesirable manner. In addition, Stava et al.’s heuristic applies only to tubular surfaces, while interactive thickening can become laborious for complex shapes.

In contrast to the above heuristics-based and interactive part thickening approaches, our method automatically computes optimal shape deformations to reduce mechanical stress. Our optimization does not make any particular assumptions about the geometry of vulnerable parts. It does not restrict deformations to a particular type. It lets the optimization to perform shape corrections under the constraint that the user’s design should be as intact as possible.

Shape optimization approaches have been demonstrated in the case of improving the balance [101], spinnability [16], and in general, aggregate mass properties of objects [94]. These methods use either cage-based deformations [101, 16] or displacements with offset surfaces [94]. Our approach instead aims to reduce mechanical stress. Reducing mechan-
Figure 4.4: A comparison between our method and local thickening. The input shape is a utility hook. We use each of our method and local thickening to generate an output that can withstand the same maximum force load. Note how our result is better at preserving shape features. In addition, the volume of our result is 20% less than local thickening, thus our method is more efficient for 3D printing as well.

Mechanical stress often requires localized changes, sparsely applied throughout the input object, which cannot be captured well by the above deformation techniques.

Shape optimization approaches for stress reduction have been studied in the mechanical engineering literature. However, these are often restricted to the case of parametric surfaces [51, 125, 19], such as isogeometric analysis which uses B-splines or Bezier patches, level set representations [10, 9, 41], or procedural models based on high-level design variables [20, 104, 118]. Parametric surface representations impose limitations on the surface topology of input shapes, while level set methods tend to lose surface detail and are applied to low-resolution grids due to their high computational cost. Procedural models handle only particular families or types of shapes i.e., those that can be created through given design parameters. Our method instead directly optimizes mesh representations, which are the most common representations in shape design. Mesh optimization approaches have been investigated in mechanical engineering [84, 69, 55], but require filtering (e.g., Laplacian smoothing) and other forms of post-processing at each optimization step to combat instabilities and surface noise. As a result, in these methods, surface detail and features are lost, or con-
vergence to a plausible solution is not guaranteed. In contrast to all the above mechanical engineering techniques, our method combines physics simulation and feature-preserving shape deformation into one optimization problem efficiently solved through Netwon-based techniques. Our method is also designed to preserve surface features and smoothness, which are highly desired properties in shape design.

Homogenization methods have also been proposed to optimize the topology of the shape [21, 115]. In such methods, the shape is discretized into a grid, and micro-structures are assigned to the grid cells. Such micro-structures are usually undesirable due to the difficulty in manufacturing. Post-processing is required if one needs to eliminate such micro-structures [8]. As with level-set methods, homogenization methods lose surface details in the optimized shape. Also, surface features are also not honored. In comparison, although our method cannot perform topology optimization, we preserve design features of the shape, and do not have the manufacturing problems with micro-structures.

4.3 Overview

Our goal is to optimize the shape of a 3D object so that it can withstand forces in common use scenarios. For example, given an input chair or table (Figure 4.1) along with its material properties (e.g., thermoplastic), boundary conditions (e.g., standing on the floor), and external forces exerting on specified regions of the surface, we optimize the shape to ensure that it will not yield under that load. An object yields when the mechanical stress induced by the loading exceeds its capacity to resist the load. Thus, we aim to reduce the mechanical stress of the object so that it is lower than the yield strength of its material. Given the force load, the mechanical stress is a function of the material properties of the object and its shape. We assume the material properties are pre-specified and determined by the printing material. In addition, we assume the input shape is represented by a volumetric mesh. If the input shape is a surface mesh, we tetrahedralize it to obtain its volumetric version.
A naive solution to this problem would be to exhaustively try different perturbations of the input mesh, and for each perturbed shape, use physics simulation to compute the shape deformation and stresses caused by the load. Then numerical gradients over the mesh points could be computed and subsequently gradient descent could be used to reduce stress at points exceeding the material’s yield strength. Such approach would be prohibitively slow, extremely prone to numerical instabilities, and lead to rather unpredictable results.

Instead, we integrate the physics simulation into the shape optimization, which enables the computation of analytic gradients. To begin, given an input volumetric mesh $X^0$ with vertex coordinates $\{x^0_1, x^0_2, \ldots, x^0_M\}$, our optimization method solves for: (i) vertex coordinates of the optimized shape $X = \{x_1, x_2, \ldots, x_M\}$ (called material coordinates or rest state in the context of physics simulation), and (ii) vertex coordinates of the deformed shape $P = \{p_1, p_2, \ldots, p_M\}$ caused by the load (called world coordinates or deformed state). To ensure that the object will not yield, we introduce inequality constraints in formulation, expressing that the stress at each tetrahedron $t$ of the volumetric mesh should be lower than the yield strength $C$ of the material. Specifically, we use the von Mises stress $\hat{\sigma}_t$ per element, which is a widely employed measure of stress in computer graphics and mechanical engineering [86] for checking whether an object yields under a given load. The stress depends on the deformation of the object under the load, thus, as part of our optimization problem, we need to solve for the deformed state (world coordinates) of the object when it reaches a force equilibrium. We use equality constraints to express that the deformed state must reach the force equilibrium under boundary conditions. Boundary conditions involve boundary vertices $v \in B$ that remain fixed during deformation (e.g., vertices at the bottom of table legs touching the floor). For these boundary vertices, we use equality constraints to express that they should remain fixed. For all other vertices $v \notin B$, we use equality constraints to express that the sum of both internal and external forces $f_v$ should be zero.

In addition to the above simulation constraints, we include additional equality constraints to preserve important geometric properties related to the aesthetics and function-
ality of the input shape, as well as to ensure numerical stability during the optimization. These are called geometric constraints, which include: (i) symmetry constraints to ensure that the optimized shape will preserve local and global symmetries found in the original shape (e.g., reflective and rotational symmetries); (ii) interior uniformity constraints to ensure that the finite elements (tetrahedra) remain well-formed and all interior mesh vertices remain uniformly spaced during the optimization, a necessary condition for numerical stability and accuracy; (iii) user-defined constraints that practically ensure that the object meet the design constraints (e.g., the height of the table should be fixed) after optimization. These user-defined geometric constraints are optionally given as part of the input to our algorithm, and depend on the user’s preferences on how the object will be used.

Finally, we use an objective function to penalize geometric deviation of the optimized shape from the original shape. Ideally, the surface of the optimized shape should remain as similar as possible to the original shape. Thus, our objective function is designed to minimize changes to the original surface, preserve its original details and smoothness.

Mathematically, our optimization problem is defined as follows:

\[
\arg \min_{\mathbf{X}} D(\mathbf{X}, \mathbf{X}^0) \quad \text{(Objective function)}
\]

subject to:

\[
\forall v \in B: \mathbf{x}_v = \mathbf{p}_v, \quad \forall v \notin B: \mathbf{f}_v(\mathbf{X}, \mathbf{P}) = 0 \quad \text{(Simulation constraints)}
\]

\[
\forall t: \hat{\sigma}_t^2(\mathbf{X}, \mathbf{P}) < C^2 \quad \text{(Stress constraints)}
\]

\[
g(\mathbf{X}, \mathbf{X}^0) = 0 \quad \text{(Geometric constraints)}
\]

where \(v\) is a mesh vertex, \(t\) is a tetrahedron, \(D(\mathbf{X}, \mathbf{X}^0)\) is our objective function expressing the geometric dissimilarity between the optimized shape \(\mathbf{X}\) and original shape \(\mathbf{X}^0\), \(\hat{\sigma}_t\)
is the von Mises stress at a tetrahedron $t$ (expressed as a function of the optimized shape and its deformed state $P$), $f_v$ is the sum of internal and external forces on a vertex $v$, and $g(X, X^0)$ is a function encoding the geometric relationships of points in the original and optimized shape. Notice that our problem formulation is also known as simultaneous analysis and design formulation (SAND, [14]) in the mechanical engineering literature. In the following section, we discuss how the above functions are computed, and our algorithm for solving the optimization problem.

4.4 Shape Optimization

4.4.1 Elasticity Model

The forces per vertex consist of external and internal forces. The external forces are part of the input – they include gravity and force exerted on surface areas according to a typical use scenario for the object. The internal forces are elastic forces resulting from the deformation of the object – they are a function of the vertex coordinates of the undeformed shape $X$ (material coordinates) and its deformed version $P$ (world coordinates). Since the input shape is discretized into finite elements (tetrahedra), the deformation is measured per tetrahedron $t$. Consequently, the total force per vertex $v$ can be expressed as:

$$f_v(X, P) = \sum_{\forall t, v \in t} f_{t,v}(X, P) + f_{ext}^v$$  \hspace{1cm} (4.19)

where $f_{ext}^v$ are the external forces per vertex, and $f_{t,v}(X, P)$ expresses the internal elastic forces per vertex of each tetrahedron $t$ as a function of the optimized and deformed shape. As commonly done in structural analysis methods, we assume linear elasticity to model the elastic forces in our object. The elastic forces are modeled as the partial derivatives of the strain energy per tetrahedron with respect to its deformed vertex coordinates [102]:

$$f_{t,v}(X, P) = \frac{\partial U_t(X, P)}{\partial p_v}$$  \hspace{1cm} (4.20)
The strain energy per tetrahedron $t$ is in turn a function of its volume $V_t$, its deformation expressed through its strain tensor $\epsilon_t$ (symmetric rank-two tensor i.e., $3 \times 3$ matrix), and the material properties $E$ of the object (rank four tensor), which are given as part of the input. In all our experiments, we assume isotropic material. In this case, the material properties tensor depends only on the Young’s modulus, which measures the object’s stiffness, and Poisson’s ratio, which measures the object’s contraction when it is stretched. The strain energy is expressed as follows [102]:

$$U_t(X, P) = \frac{V_t}{2} \epsilon_t \otimes E \epsilon_t$$ (4.21)

where $\otimes$ refers to tensor multiplication. Here, the strain tensor is a function of the optimized and undeformed shape, while the volume of the tetrahedron is a function of the vertex coordinates of the optimized shape. Specifically, if $v_0, v_1, v_2, v_3$ are the four vertices of a tetrahedron and $\bar{X}_t$ represents its edge vectors stored in a $3 \times 3$ matrix $\bar{X}_t = [x_{v_1} - x_{v_0}, x_{v_2} - x_{v_0}, x_{v_3} - x_{v_0}]$, the volume is simply determined as: $V_t(X) = \frac{1}{6} |\bar{X}_t|$, where $|\cdot|$ refers to the determinant of the matrix. The strain is a function of the deformation gradient per tetrahedron, which describes its scaling and rotation due to the compression or elongation of the object’s material. By representing the edge vectors for each deformed tetrahedron in a $3 \times 3$ matrix $\bar{P}_t = [p_{v_1} - p_{v_0}, p_{v_2} - p_{v_0}, p_{v_3} - p_{v_0}]$, the deformation gradient is computed as:

$$F_t(X, P) = \bar{P}_t \bar{X}_t^{-1}$$ (4.22)

To compute the strain tensor, we follow the invertible finite element methods [57] to extract the scaling from the deformation gradient using SVD: $F_t = U_t \Sigma_t V_t^T$. During each iteration of the optimization, we assume that each tetrahedron undergoes an infinitely small rotation. This assumption works well in practice. Since the rotation matrices $U$ and $V$ are constant during each optimization step, only the scaling matrix is a function of the optimized and deformed shape. We use the linear Cauchy-Green strain tensor to express
the strain based on the scaling per tetrahedron: \( \epsilon_t = \Sigma_t - I \), where \( I \) is the identity matrix.

Based on the SVD of the deformation gradient and Equation 4.22, we can express the strain tensor as a function of the optimized and deformed shape:

\[
\epsilon_t(X, P) = U_t^T \bar{P}_t \bar{X}_t^{-1} V_t - I
\]  

(4.23)

Replacing the strain tensor function in Equation 4.20, the force function can be rewritten as:

\[
f_{t,v}(X, P) = \frac{\partial U_t(X, P)}{\partial p_v} = V_t \left( U_t^T \bar{P}_t \bar{X}_t^{-1} V_t - I \right) \otimes E \frac{\partial \epsilon_t}{\partial p_v}
\]  

(4.24)

where the derivative of the strain tensor \( \epsilon_t \) for a tetrahedron with respect to each of the three coordinates of a deformed vertex \( p_v \) at that tetrahedron are the following:

\[
\frac{\partial \epsilon_t}{\partial p_{v,k}} = U_t^T \frac{\partial \bar{P}_t}{\partial p_{v,k}} \bar{X}_t^{-1} V_t
\]  

(4.25)

Here \( k \) is an index for the \( x, y \) or \( z \) coordinate in the vertex \( p_v \). The partial derivative of each element \((i, j)\) in the matrix \( \bar{P}_t \) storing the deformed edges are the following:

\[
\frac{\partial \bar{P}_t(i, j)}{\partial p_{v,k}} = \begin{cases} 
1 & \text{if } v = v_j \text{ and } k = i \\
-1 & \text{if } v = v_0 \text{ and } k = i \\
0 & \text{otherwise}
\end{cases}
\]  

(4.26)

To solve our optimization problem, we also need analytic gradients of the force function. We refer the reader to the Appendix B and C for its analytic gradients.

### 4.4.2 Stress Function

As described in Section 4.3, during the optimization, we set inequality constraints to ensure that the von Mises stress in the object will not exceed the material’s yield strength.
The von Mises stress is extracted from the stress tensor that characterizes the state of stress at each tetrahedron. The stress tensor is a symmetric $3 \times 3$ matrix:

$$
\sigma_t = \begin{bmatrix}
\sigma_{11} & \sigma_{12} & \sigma_{13} \\
\sigma_{21} & \sigma_{22} & \sigma_{23} \\
\sigma_{31} & \sigma_{32} & \sigma_{33}
\end{bmatrix}
$$

(4.27)

Given the strain tensor $\epsilon_t$ of a tetrahedron $t$, the stress tensor is expressed as $\sigma_t = E \epsilon_t$. Since the strain tensor is a function of the undeformed shape and its deformed version (Equation 4.23), the stress tensor elements are also functions of their vertex coordinates. The von Mises stress is computed from the stress tensor as follows [86]:

$$
\hat{\sigma}_t^2(X, P) = 0.5[(\sigma_{11} - \sigma_{22})^2 + (\sigma_{22} - \sigma_{33})^2 + (\sigma_{33} - \sigma_{11})^2
+ 6(\sigma_{23}^2 + \sigma_{31}^2 + \sigma_{12}^2)]
$$

(4.28)

Our optimization procedure requires the analytic gradient and Hessian of the above function, which we provide in the Appendix.

4.4.3 Geometric Constraints

We now describe the geometric constraints that we use to preserve important geometric properties of the input shape related to its aesthetics, functionality and numerical stability during optimization.

Symmetry constraints. Symmetry is an important factor in the aesthetics of man-made objects. Ignoring symmetry can cause a shape to become asymmetric during optimization, which is undesirable (Figure 4.5). To incorporate symmetry, we first detect dominant planar reflective and rotational symmetries in the original shape [89]. We aim to preserve these symmetries in the optimized shape. A planar reflective symmetry constraint can be defined through a reflection matrix $S_m$. For each pair of symmetric points $x_i$ and $x_j$ in the
optimized shape, we set $S_m x_i = x_j$ and $S_m x_j = x_i$. A rotational symmetry is similarly defined through a rotation matrix $S_r$. For a set of $k$ ordered points $\{x_1, x_2, \ldots, x_k\}$ related through a rotational symmetry, we set $S_r x_i = x_{i+1} (1 \leq i < k)$ and $S_r x_k = x_1$. Other types of symmetry can also be supported.

(a) original shape  
(b) w/o symmetry  
(c) with symmetry

Figure 4.5: Without symmetry constraints, the resulting optimized shape becomes asymmetric. This can be addressed by incorporating symmetry constraints.

**Interior uniformity constraints.** During optimization, a key factor for numerical stability and convergence is to ensure that all finite elements are well-shaped, free of self-intersections, and all interior (non-surface) vertices are uniformly spaced. Leaving the interior vertices float freely during optimization would also cause inaccuracy in the stress calculations and analytic gradients. We account for this by forcing all interior vertices of the optimized volumetric mesh to be positioned exactly at the centroid of their neighbors. Mathematically, for each interior vertex, we set its position to be:

$$x_i = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} x_j \quad (4.29)$$

where $\mathcal{N}(i)$ is the set of neighboring vertices. The above expression can be also thought of as forcing the graph laplacian of the interior vertices to be zero. This constraint is also guaranteed during initialization where we tetrahedralize the input shape.

**User-defined constraints.** Our method allows the user to specify additional geometric constraints for aesthetics or functionality purposes. The most common constraint is to
force certain edges to be straight, or certain vertices to remain fixed during optimization. An example is shown in Figure 4.6: without user-defined constraints, the top of the shape is deformed. To keep it flat and the edges straight, the top surface is explicitly marked as fixed. The result of optimization is shown in Figure 4.6c. These user-defined constraints can be achieved by setting $x_i = x_i^0$ for all vertices that need to remain fixed, where $x_i^0$ is the position of the vertex on the original shape.

![Figure 4.6](image_url)

(a) original shape  (b) w/o user constraints  (c) with user constraints

Figure 4.6: Without user-defined constraints, the resulting optimized shape in (b) has a top surface that is deformed. This can be addressed by user-defined constraints that keep the top surface fixed.

**Summary.** In general, we represent all the above geometric constraints in our formulation as equality constraints, in the form of $g(X, X^0) = 0$. Each $g$ is a linear function of the vertices on the input and optimized shapes.

### 4.4.4 Objective function

Our objective function aims to penalize surface dissimilarity between the original and the optimized shape. The goal is to keep the optimized shape as similar as possible to the original. We measure surface distance using an *extrinsic* distance as well as an *intrinsic* distance. The extrinsic distance directly computes the distance between surface vertices of the two shapes – it prevents the optimized surface to deviate significantly away from the original surface in Euclidean space. The intrinsic distance computes the distance of differential surface properties – it preserves surface details and smoothness, and prevents
Figure 4.7: Block diagram of our optimization approach.

Figure 4.8: Without extrinsic distance, the optimized shape may deviate significantly from the original (b); without intrinsic distance, the result can contain bumpy surface/jaggy edges and lose surface details (c). Including both measures is important to ensure the result is geometrically similar to the original.

the resulting surface from becoming bumpy/jaggy. An example is shown in Figure 4.8. In practice we found that both measures are important for producing an optimized shape that is geometrically similar to the original shape.

Extrinsic distance. Our extrinsic distance considers the squared Euclidean distance between the vertices of the original surface $X^0$ and the optimized surface $X$:

$$D_{\text{extrinsic}}(X, X^0) = \sum_{i \in S} \| x_i^0 - x_i \|^2$$  \hspace{1cm} (4.30)

where $S$ is the set of surface vertices. We do not consider the interior vertices as they are not related to shape appearance. Note that during optimization the number of surface vertices does not change, and we have an explicit correspondence between the original and optimized surface vertices, thus this distance is easy to compute.
**Intrinsic distance.** Our intrinsic distance compares the differential surface properties in terms of surface Laplacians. Surface Laplacians have been extensively used in detail-preserving shape deformations due to their ability to capture information about the local shape of the surface, and in particular the size and orientation of local surface details [112, 113]. The Laplacian of a surface vertex can be computed as follows:

$$L(x_i) = \sum_{j \in \mathcal{N}(i)} \omega_{i,j} (x_i - x_j)$$  \hspace{1cm} (4.31)

where $\mathcal{N}(i)$ here represents the neighboring vertices on the surface (we exclude any interior vertices from the calculation of surface Laplacians), and $\omega_{i,j}$ are weights controlling the desired properties for the Laplacian [128]. We experimented with both uniform weights ($\omega_{i,j} = 1/|\mathcal{N}(i)|$) and the cotangent scheme [85] and in practice did not observe any significant differences of the two, due to the relatively uniform tesselations of our input meshes. Thus for simplicity, we use uniform coefficients.

Following [112], our intrinsic distance computes differences between transformed Laplacians of the original surface vertices and Laplacians of the corresponding surface vertices in the optimized shape:

$$D_{\text{intrinsic}}(X, X^0) = \sum_{i \in \mathcal{S}} \| T_i L(x_i^0) - L(x_i) \|^2$$  \hspace{1cm} (4.32)

where $T_i$ is a transformation matrix that optimally translates, rotates, and scales the original surface Laplacian of each vertex $i$ before comparing it to its surface Laplacian on the optimized surface. The reason for transforming the surface Laplacian is that we want to allow local surface rotations and scaling since they can both contribute to reducing the mechanical stress of the object without significantly affecting its surface appearance. Following [112], the transformations can be computed by aligning each surface neighborhood in the original shape with the one on the optimized shape in least-squares sense:
The transformation matrices are internal variables in our algorithm and are solved simultaneously with the optimized shape $X$. We experimented with transformations restricted to rotation, translation and uniform scaling (as in [112]), but also with affine transformations. In the case of affine transformations, we compute them using the 2-ring surface vertex neighborhoods in the above expression to ensure enough number of constraints for the least-squares solution. In the case of planar neighborhoods on the original surface, the least-squares solution can still be ill-defined. Please read the background (Section 4.1.2) for more details.

One consequence of allowing affine transformations is that under significant deformation it can lead to shearing distortions [46]. However, in our case, due to the extrinsic distance and overall optimization, surface changes are generally moderate. In addition, non-uniform local stretching can be particularly effective for reducing mechanical stress in man-made objects without significant impact on the surface appearance. Therefore we adopt affine transformations by default and use those in our experiments. If needed, users can easily override this option and execute the shape optimization by restricting the transformations to rotations and uniform scaling only, as in [112].

**Objective function.** Our objective function is defined as a weighted sum of the intrinsic and extrinsic distances:

$$ D(X, X^0) = w_1 D_{intrinsic}(X, X^0) + w_2 D_{extrinsic}(X, X^0) $$

where $w_1$ and $w_2$ are the weights. By default, we set $w_1 = 10^5$ and $w_2 = 1$, which have worked well in our tests. Reducing $w_1$ tends to cause loss of surface details and surface smoothness, while reducing $w_2$ can cause the optimized surface to deviate significantly from the original.
4.4.5 Optimization approach

We now describe the procedure to solve our optimization problem. Although our objective function has the desirable property to be a quadratic (i.e., convex) function on the shape $X$ we wish to estimate, the force and von Mises stress functions involved in the constraints are non-linear and non-convex over both the shape $X$ and its deformed version $P$. In particular, any small change in $X$ or $P$ can easily cause a large violation of the force equilibrium constraints. An additional challenge is that for shapes of reasonably high resolution, the number of unknown variables is on the order of tens of thousands at least. Specifically, the number of unknown variables is equal to the number of vertices of the optimized shape, multiplied by a factor of 6, since their deformed state is included in the unknowns, and we deal with 3D coordinates. Thus, to solve our problem, we need a non-linear and large-scale optimization method. An important fact that we can leverage is that all functions are twice differentiable, and we are able to derive both analytic gradients and Hessians. We experimented with various solvers including Sequential Quadratic Programming, interior-point, and penalty methods [95]. None of the techniques worked out of the box. In the end we implemented our own variant of the penalty method [82] which we describe below.

The original penalty method reformulates a constrained optimization problem as a sequence of unconstrained problems. It has the advantage of fast local convergence guarantees under relatively weak assumptions [25]. The unconstrained problems are formed by adding a term for each equality and inequality constraint, called penalty function, to the objective function. Each penalty function consists of a penalty coefficient multiplied by a measure of violation of each constraint. The violation measure is non-zero in regions where the constraint is violated and is zero where the constraint is met. By iteratively increasing the penalty coefficients, the solver converges to a solution where the constraints are satisfied.

Our approach is inspired by the penalty method. Specifically, we use penalty functions to handle the inequality constraints for the von Mises stress. However, we found that our
equality constraints cannot be handled well by penalty functions, as discussed below. Thus in our case, we reformulate our equality- and inequality- constrained optimization problem, presented in Section 4.3, as a sequence of equality-constrained problems as follows:

\[
\arg \min_{X} D(X, X^0) + \delta \cdot \sum_{t} h(C_t^2 - \hat{\sigma}_t(X, P)) \quad \text{(New objective function)}
\]

subject to:

\[
\forall v \in B: x_v = p_v, \quad \forall v \notin B: f_v(X, P) = 0 \quad \text{(Simulation constraints)}
\]

\[
g(X, X^0) = 0 \quad \text{(Geometric constraints)}
\]

where \( h(\cdot) = \min(0, \cdot)^2 \) is the penalty function that measures the degree of violation in the inequality constraints, and \( \delta \) is a penalty coefficient. Starting from a small value (\( \delta = 0.01 \)), we solve the above optimization problem and check whether our inequality constraints are satisfied. If they are not satisfied, the penalty coefficient is increased by a factor of 2, and we solve the problem again starting with the solution found in the previous iteration. Solutions of the successive optimization problems will eventually converge to the solution of the original problem. Practically, in our experiments, this happens when \( \delta \) reaches the value of 100 at most.

In each iteration, we still need to still solve an equality-constrained optimization problem. Although we could use a penalty function (e.g. a quadratic function) again for those, thus convert the problem to a completely unconstrained one (as done in the original penalty method), this approach did not work. The reason was that the force equilibrium constraints could not be satisfied without causing the optimization to become ill-conditioned due to extremely large penalty coefficients. We note that an augmented Lagrangian approach [25] could deal with this problem, but at a higher computational cost. We instead resort to
a simpler approach, by solving the equality-constrained optimization problem using the Newton’s approach [24]. Mathematically, it is represented as (this is also known as the Karush–Kuhn–Tucker, or the KKT system):

\[
\begin{bmatrix}
H & J^T \\
J & 0
\end{bmatrix}
\begin{bmatrix}
\{\Delta X, \Delta P\} \\
w
\end{bmatrix}
= -
\begin{bmatrix}
g \\
b
\end{bmatrix}
\]  

(4.35)

where \(H\) and \(g\) are the Hessian and gradient of the new objective function respectively, \(J\) is the Jacobian of the equality constraints, and \(b\) represents the function values of the constraints. We solve the above system for multiple iterations until convergence: at each iteration, the shape \(X\) is updated by \(t\Delta X\), and similarly its deformed version is also updated by \(t\Delta P\), where \(t\) is a step size determined by a bisection line search. In practice, to ensure that the deformed state of an optimized shape reaches the force equilibrium (i.e., satisfy the simulation constraints), for each update in the shape \(X\), we perform several more updates to its deformed state \(P\) until the force equilibrium is reached (sum of internal and external forces per vertex is practically zero). Note that the system is sparse since each vertex is only affected by its local neighbors. To this end, we apply the sparse LU decomposition algorithm available in the Eigen [50] library. We also refer the reader to our source code for our implementation.

### 4.5 Results

To understand how our algorithm works, we begin by using a simple rectangular board as the testing shape, as shown in Fig. 4.9. Here the two ends of the board are fixed at the bottom (indicated by blue color), and hence they are set as fixed boundaries to the algorithm. Notice that setting boundaries as hard constraints in this example will cause singularity, where in theory the stress on fixed points becomes unbounded. To avoid such singularity, we relax the boundary constraints to be soft constraints. Whenever boundary vertices deviate from their rest states, a spring force will pull them back. External force
is applied in the middle region (indicated by red color). Under these settings, we ran our algorithm and the results are shown in Fig. 4.10. Fig. 4.10a shows the shape before optimization, and Fig. 4.10b shows the output. Fig. 4.10c and 4.10d visualize the von Mises stress distribution before and after optimization. Red color corresponds to the highest stress and blue corresponds to the lowest stress. Our algorithm automatically strengthens

![Figure 4.9: A simple test shape](image)

the input shape in areas with high von Mises stress, and the result has significantly reduced von Mises stress. Quantitatively, the optimized shape can withstand twice as much (i.e. 100% more) force/weight as the original shape.

![Figure 4.10: Results of the simple test shape. Right: color coding of the von Mises stress.](image)

Fig. 4.11 shows a gallery of experimental results including furniture and utility tools. For each input shape we show the region and direction of the external force, and the sur-
Figure 4.11: A gallery of experimental results. The first row shows the original shapes and region/direction that external forces are applied. We find the maximum force each original shape can withstand, and apply our algorithm to optimize the shape until it can withstand 50%, 100%, and 200% additional force respectively. The results are shown in the next three rows. Each result contains a rendered image of the optimized shape, a silhouette image overlaying the optimized shape on the input to show the deformation, and the color encoding of the vertex displacement (i.e. how far the vertex has moved before and after optimization) as well as von Mises stress distribution over the surface.
face/boundary that’s fixed. We find the maximum force each shape can withstand (i.e. when the von Mises stress reaches the material limit). We then increase that force by 50%, 100%, and 200% respectively, and apply our algorithm to optimize the shape until it can withstand that much additional force. For each optimized shape, we show a rendering of the shape, a silhouette image that compares the difference between the optimized shape with the original shape, and two color-coded images indicating the displacement of vertices after optimization (i.e. how far each vertex has moved before and after optimization), and the von Mises stress distribution. Observe that as we increase the external force, the optimized shape changes more significantly, although remaining similar to the original shape. A number of the examples, such as the utility hook and the d-shaped chair, undergo global deformations, which cannot be easily achieved using techniques based on local thickening.

**Objective function.** We compare the effects of different distance measures in our objective function in Figure 4.8. Using only the intrinsic distance measure, the result tends to have large global deformations and the surface deviates significantly from the original shape (Fig. 4.8b). Using only the extrinsic distance measure, the surface tends to become noisy and bumpy (Fig. 4.8c). Our objective function combines both extrinsic and intrinsic distance measures, producing satisfactory results (Figure 4.8d).

**Constraints.** In Figure 4.5 we show the effect of symmetry constraints. Our algorithm automatically detects symmetries in the original shape, and assign them as geometric constraints. The result observes symmetries in the original shape, which is important for both aesthetics reasons and for preserving the shape appearance. In Figure 4.6 we show an example of custom constraints defined by the user. Here the result shown in 4.6(b) has a top surface that’s curved. This is not ideal as the top of a stool is typically flat. By adding custom constraints, the optimizer can keep the top surface flat during the optimization, thus the result matches the intended application scenarios of the object.

**Physical validation.** We perform physical validations by using a professional force gauge (Instron Universal Testing machine), which applies increasing axial force on the shape until
it breaks. We have printed each original shape and the optimized result using a 3D printer (CraftBot) with PLA material, 100% infill, and 100 micron resolution. Figure 4.12 shows two photographs of the testing machine.

Figure 4.13 shows our test results for two sets of models: the twist-leg table, and the four-leg stool. We use our algorithm to compute an optimized shape that can withstand twice (100% more) force than the maximum force allowed on the original shape. Because the optimized shape is typically slightly than the original shape, for fair comparison, we normalize volume of the optimized shape with the original shape before printing, so that they have the same weight. For each example we plot the force measured on the original shape (gray curve) and that on the optimized shape (purple curve). The point where each curve reaches the bottom is where the model breaks.

From the measured data, for the twist-leg table, the original shape breaks at $282N$ and the optimized shape breaks at $425N$. So the optimize shape can withstand 50.7% more force than the original shape. This is worse than simulation (which predicts 100% more force) due to a variety of reasons such as normalization of volume (which will increase the maximum force that the original shape can withstand), printing quality, theoretical vs. practical material parameters, and plastic deformations which is not considered by our formulation. For the four-leg stool model, the original shape breaks at $523N$ and the optimized shape breaks at $1605N$. So the optimized shape can withstand 207% more force. This is considerably better than simulation.

**Timings.** The running time for shape optimization in typical force scenarios ranges from 2 hours for our model with fewest number of vertices ($1K$ vertices) to 6 hours for our largest model ($3K$ vertices). The running times are comparable to the time needed to print a 3D model. Our implementation is currently far from optimal, and we believe that the running times can be significantly improved with a GPU-based implementation to solve the KKT system.
Figure 4.12: Photographs showing the Instron Universal Testing machine, which applies increasing axial force on the shape until it breaks. It can accurately measure the force applied on the testing shape, and the distance that the load cell has moved (head extension). We use the machine to perform physical validations. The picture on the right shows a close-up view of the object being tested.

Figure 4.13: Physics validation results for the twist-leg table and the four-leg stool models. In each graph, the x-axis is the head extension in millimeter (i.e. the distance that the load cell has moved); the y-axis is the measured force in kilonewton. The original shape is plotted in gray, and optimized shape plotted in purple. The original and optimized shapes are normalized before printing, so they have equal volume/weight. The point where the curve reaches the bottom is where it breaks under the corresponding force.
4.6 Blue Noise Sampling using an SPH–based Method

<table>
<thead>
<tr>
<th>Density Diff.</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>25</th>
<th>100</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations Time (s)</td>
<td>373</td>
<td>344</td>
<td>300</td>
<td>288</td>
<td>217</td>
<td>228</td>
</tr>
<tr>
<td>Time (s)</td>
<td>1.108</td>
<td>0.936</td>
<td>0.842</td>
<td>0.780</td>
<td>0.593</td>
<td>0.640</td>
</tr>
</tbody>
</table>

Table 4.1: This table shows samples generate using our algorithm, by varying the ‘density difference’ parameter. For each example, we show the Fourier power spectrum, associated radial means and anisotropy of the spectrum. From left to right we observe that the spectrum transitions from low effective Nyquist Frequency and oscillation to high effective Nyquist Frequency and oscillation. Our sampling method runs extremely fast. It generates various blue noise with 10000 points per second.

As is discussed previously, one issue that we face in the physics-based optimization project is that for general shapes, it might be difficult to tetrahedralize them uniformly due to small features and non-uniformity of surface triangles. Thus, in this section I am going to introduce a blue noise sampling method based on Smoothed Particla Hydrodynamic (SPH), which might provide a solution to this problem. Such method can uniformly sample a 3D mesh while nicely preserving surface feature. This is done by generating feature-aligned blue noise samples within a volume.

In this section I will firstly describe how to generate blue noise samples with SPH-based method in 2D domain. The method can generate samples with tunable trade-off between effective Nyquist frequency and oscillation in the blue noise spectrum. As a result, we can generate blue noise samples with a variety of different blue noise profile, ranging from...
Lloyd’s relaxation to CCVT samples and beyond. When applied on a volumetric shape, our samples can nicely aligned with edges on mesh surface, thus it can preserve surface features nicely.

### 4.6.1 The Core Idea

SPH is an interpolation method for particle system [92]. The concept of SPH in fluid simulation is to interpolate fluid quantities at arbitrary positions and to approximate the spatial derivatives with a finite number of sample positions[56]. SPH itself is a process which forcing the density approaching the rest density to minimize the pressure difference between points [92]. This explains why in the regions with smooth flow field, SPH particles are uniformly distributed [3] — hence simulate blue noise samples.

#### The Basic Algorithm.

In SPH [91], a scalar quantity $A_i$ at location $r_i$ is approximated with a set of neighbouring points $j$ at $r_j$ using radial symmetrical smoothing kernels $W_{ij}$:

$$A_i = \sum_j \frac{m_j}{\rho_j} A_j W_{ij}$$  \hspace{0.5cm} (4.36)$$

$$W_{ij} = W(r_i - r_j, h)$$  \hspace{0.5cm} (4.37)$$

where $m_j$ is the mass of point and $\rho_j$ the density. The kernel satisfies $\int W(r, h)dr = 1$ and $W(r, h) = 0$ when $\|r\| > h$. $h$ as a smooth radius defines the neighbourhood size.

Following Equation 4.36 and presuming each point has the same mass, the point density can be represented by:

$$\rho(r) = m_j \sum_j W_{ij}$$  \hspace{0.5cm} (4.38)$$

In SPH, the relationship between the mass the density can also be written as:

$$\rho(r) = \frac{m_j}{V_i}.$$  \hspace{0.5cm} (4.39)$$
Figure 4.14: Using our SPH sampling method to simulate the Lloyd’s relaxation profile. This regular hexagonal distribution results from ensuring the convergence condition \( \sum_j \nabla W_{ij} = 0 \).

where \( V_i \) is the volume. Notice that the volume of points will be the same if the density of points is the same, matching the idea of the capacity constraint in CCVT. However, we don’t need to guarantee the same capacity (volume) in every iteration in SPH — instead the SPH algorithm automatically reaches the same capacity (density) in convergence which guarantees the quality of SPH sampling.

The pressure force is computed using spatial derivatives by applying SPH rules to the Navier–Stokes equation [48]:

\[
\mathbf{F}^{\text{press}}_i = -\frac{m_i}{\rho_i} \sum_j m_j \left( \frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W_{ij} \tag{4.40}
\]

where \( p \) is the pressure calculated directly from density:

\[
p = k(\rho - \rho_0), \tag{4.41}
\]

where \( \rho_0 \) is named as the rest density in SPH. The pressure force is created whenever there is a pressure difference and it acts in the direction from high pressure to low pressure along the negative pressure gradient.

Equation 4.40 introduces a spring–like behavior to the pressure calculation: while a density higher than the rest value produces a positive pressure that pushes the points away,
a lower density will result in a negative pressure and brings the points together. As such, the pressure term aims to equalize the density differences throughout the fluid. In an equilibrium the pressure and density will be identical everywhere, thus the same volume (capacity) is achieved. The gas constant $k$ is a damping factor.

If a point is inserted at a density center where there is no density difference, the pressure force returns to 0. The density center is actually the mass center according to the definition of density, intuitively similar to Voronoi relaxation, which moves points towards the center of the cell. In SPH the pressure force controls the pattern of sampling. This idea is consistent with Lloyd’s algorithm which converges to equal area, while SPH converges to equal density. This density calculation implicitly matches the capacity-constraint in CCVT.

When using SPH for fluid simulations a viscosity force is included to synchronize the velocity of all particles. In the sampling context the smooth dynamic flow of a simulation is irrelevant: instead we want the samples to settle with 0 velocity as quickly as possible. A simple damping coefficient is applied to the pressure force which speeds up convergence, achieving an equivalent behavior to viscosity but with considerably better performance. We use a damping coefficient of 0.5 in all our examples.

**Convergence.** We consider the sampling to have converged when the point displacement is less than a small $\epsilon$. Upon convergence, the points have the following properties:

1. All points have approximately the same density, i.e. $\forall i, j, \rho_i \approx \rho_j$. Notice that this does not imply point density reaches the rest density.

2. The pressure force of all points reaches zero, i.e. $\forall i, \mathbf{F}_{i}^{\text{pres}} = 0$.

The first property means there is no pressure difference among points and the volume of the points are the same (surface area in 2D).
Figure 4.15: Using our SPH sampling method to simulate the CCVT profile. This distribution results from the convergence condition that $\rho_i = \rho_0$.

Figure 4.16: Sampling the interior and boundary of a 2D bunny shape using our SPH-based method. The color visualizes the density value at each point during the iterations. Red means high density, blue means low density, and purple in between. The boundary samples are important as they can influence the interior samples. The rightmost image shows converged result. Note the distribution quality of the resulting samples.

Combining this with Equations 4.41 and 4.40, under the setting of constant point mass, the second property can be written as:

$$F_i^{\text{pres}} = 0 \approx -\frac{2m^2k}{\rho_i^3} \sum_j \nabla W_{ij}$$

(4.42)

The above equation implies that upon convergence, for any point $i$, one of the two following conditions must stand:

1. $\rho_i = \rho_0$;

2. $\sum_j \nabla W_{ij} = 0$. 

99
Notice that in practice those two conditions are not strictly satisfied since in implementation the algorithm stops when point movement is small enough. There is a trade off between those two conditions, and its effect will be discussed in Section 4.6.3. When the first condition is satisfied for all points with a smooth kernel, one can assume that the density is approximately the same in the whole domain, or:

\[ \forall r, \quad m \sum_j W(r - r_j, h) \approx \rho_0 \]  \hfill (4.43)

The condition of \( \sum_j \nabla W_{ij} = 0 \) actually asks for all the forces acting on a sample to be symmetrical, the points in this case will favor a regular distribution — hexagonal pattern. The results are very similar with the full-converged Lloyd’s relaxation, shown in Figure 4.14. To avoid the regularity artifacts, we meet \( \rho_i = \rho_0 \) rather than \( \sum_j \nabla W_{ij} = 0 \).

When SPH reaches the stable state in a free–surface fluid simulation, the pressure tends towards 0. In sampling context, the user controls the number of the points, so it cannot be guaranteed that the density equals to the initial rest density. Therefore when the distribution converges the difference between the density of sample and the rest density will approximate a constant which may not be 0 — points are actually under high pressure. If one assigns the rest density close to the actual density of all points, most points will quickly converge with \( \rho_i = \rho_0 \) rather than \( \sum_j \nabla W_{ij} = 0 \). In this case, points will have more possibility to form tetragons, heptagons or pentagons, which achieves the similar results with CCVT, as demonstrated in Figure 4.15.

**Implementation.** We initialize the points as white noise within a given boundary. Each sample is modeled as an SPH particle, which carries associated properties, e.g. position, velocity, density and pressure. We run SPH over all the points, with the acceleration of the points obeying Newton’s third law \( a_i = \mathbf{F}_{\text{pres}}^i / m_i \).

Note that in the sampling context, gravity is not needed. Figure 4.16 shows several iterations of the SPH sampling process within a 2D bunny shape.
Discussion. The choice of kernel function affects the accuracy of the summation [56]. For the sampling example in Section 4.6.1, we use the standard kernels of Müller et al. [92] and the pressure forces of Monaghan [91] as a demonstration. The use of alternative kernels is discussed in Section 4.6.3.

4.6.2 SPH Sampling and Implementation

We initialize the points as white noise within a given boundary. Each sample is modeled as an SPH particle, which carries associated properties, e.g. position, velocity, density and pressure. We run SPH over all the points, with the acceleration of the points obeying Newton’s third law $a_i = \frac{F_{\text{pres}}}{m_i}$. Figure 4.17 shows the SPH sampling within a 2D bunny shape. Note that in the sampling context, gravity is not needed.

Surface and Volume Sampling. In SPH, points are relocated to achieve the same density within the sampling region. However, due to the insufficient sampling along the object boundary the points on the boundary become highly disordered, as shown in Fig. 4.18a.

Correction Force. The net forces push the points along the boundary in the direction of the boundary normal. We therefore need a correction force which allows the boundary points to relax onto boundary without being pushed outwards. To allow points to move freely along the boundary, we can counteract the force in the normal direction, only keeping the force along the tangential direction:

$$F_{\text{surf}}^i = -n(n \cdot F_{\text{pres}}^i)$$  \hspace{1cm} (4.44)

This correction force ensures that points near the boundary remain on the boundary of the object, leading to a better quality surface sampling. This principle is also applied to the velocity of points on the boundary, counteracting the perpendicular component of the velocity.
Figure 4.17: Sampling the interior and boundary of a 2D bunny shape using our SPH-based method. The color visualizes the density value at each point, from high (red) to low (blue).

**Normal Calculation.** Due to the insufficient sampling around the boundary, density changes dramatically for points on the boundary. Therefore we can calculate the normal for the surface points from the gradient of the density [92]:

\[
\mathbf{n} = -\sum_j \frac{m_j}{\rho_j} \nabla W_{ij}
\]  

(4.45)

The resulting normal points from the high density to the low density, out the volume.

Figure 4.18: In a the boundary points are irregularly spaced after sampling. In b, the points are more relaxed after adding the correction force, but the sampling rate is much denser than the interior. In c both correction and cohesion forces are applied, and the resulting boundary samples are coherent with the interior samples.

**Cohesion Force.** As we do not adjust density of boundary points, the boundary will attract more points to compensate the density loss, leading to a more dense sampling on the
surface (shown in Figure 4.18b). With only the correction force, once points are on the surface they cannot leave – potentially causing discontinuities between surface points and interior points. For this reason we introduce an additional force which adjusts the density difference between surface points and interior points.

We use the surface tension model of Akinci et al. [5] which contains cohesion and surface area minimization terms. As our sampling model defines the surface area beforehand (unlike free surface fluid), the surface area minimization term does not affect our results. We only apply the cohesion force:

$$F_{i}^{cohe} = -\gamma m_i m_j K_{ij} W_{cohe}(|x_i - x_j|) \frac{x_i - x_j}{|x_i - x_j|}$$ \hspace{1cm} (4.46)

where $\gamma$ is surface tension coefficient, $W_{cohe}$ is the spline function defined by Akinci et al. [5], and $K_{ij}$ is a symmetric correction factor defined as $K_{ij} = \frac{2\rho_0}{\rho_i + \rho_j}$.

This cohesion force is applied to both interior and surface points. For interior points the cohesion force is almost the same everywhere as it converges. Surface points with smaller density have larger $K_{ij}$ due to the lack of neighboring samples, which amplifies the cohesion forces of boundary points. The effect of this force is demonstrated in Fig. 4.18c. Note also that the corrected surface sampling refines the points near the boundary which have boundary points as neighbors.

**Discussion.** Our method for surface and volume sampling ensures that the surface points are exactly on the surface and also ensures the consistency between surface and volume sampling. Most boundary correction methods in SPH do not guarantee the particles to converge exactly on a fixed boundary, e.g., [6, 5], therefore we introduce the correction force. Schechter et al. [105] sampled both the interior and the surface, using an initial surface sampling as seeds to sample the volume. Our method solves both surface and volume sampling at the same time, reducing algorithmic complexity and computation overhead. Unlike the density of the ghost particles in Schechter and Bridson [105], the density of the surface samples in our method does not need extra treatment, since they only receive forces.
Input: Initial Random Sample Set $P_0$, Boundary Level set $L$, Sampling number $N$, count $i = 0$
Output: Blue noise samples $P$
while not converged do
  for $i = 0$ to $N$ do
    calculate $\rho$;
  end
  for $i = 0$ to $N$ do
    $x, v \leftarrow$ the position and velocity of current point
    if $x$ is on the surface then
      $n \leftarrow$ calculate normal
      $a \leftarrow F^{pres} + F^{cohe} + F^{surf}$
    else
      $a \leftarrow F^{pres} + F^{cohe}$
    end
    Integrate $a$ to get velocity $v'$ and position $x'$
    Apply velocity damping $v' \leftarrow \delta v'$
    if $\|x' - x\| > \epsilon$ then
      if $x'$ is within $L$ then
        $x \leftarrow x'$
        $v \leftarrow v'$
      else
        $v \leftarrow v' - n(n \cdot v')$
      end
    end
  end
end

Algorithm 1: The SPH sampling algorithm for interior and boundary points.

Surface Sampling. Our method is also suitable for surface sampling without the interior points. We achieve that by only using the correction force and replace the distance metric with geodesic distance.

Samples are initialized on the surface. We then run SPH for all surface samples, calculating $F^{surf}$ for each sample. Instead of restricting the points within the object, we constrain the points to only move on the surface. Therefore within every time step of the SPH process we need to map the samples back onto the surface. To calculate the geodesic distance, we
**Input:** Initial Random Sample Set \( P_o \), Boundary Level set \( L \), Sampling number \( N \), count \( i = 0 \)

**Output:** Blue noise samples \( P \)

**while** not converged **do**

**for** \( i = 0 \) **to** \( N \) **do**
  - calculate \( \rho \);
**end**

**for** \( i = 0 \) **to** \( N \) **do**
  - \( x, v \leftarrow \) the position and velocity of current point
  - \( n \leftarrow \) calculate normal
  - \( a \leftarrow F_{\text{pres}} + F_{\text{surf}} \)
  - Integrate \( a \) to get position \( x' \) and velocity \( v' \)
  - Apply velocity damping \( v' \leftarrow \delta v' \)
  - Map \( x' \) to the surface position \( x'_{\text{surf}} \)
  - **if** \( \| x'_{\text{surf}} - x \| > \epsilon \) **then**
    - \( x \leftarrow x'_{\text{surf}} \)
    - \( v \leftarrow v' \)
**end**

**end**

**Algorithm 2:** The SPH sampling algorithm for surfaces.

apply the light-weight algorithm of Bowers et al. [27], which approximates the geodesic distance using normal information. The normal is calculated according to the distance field. The full algorithm is shown in Algorithm 2.

As we will discussed in Section 4.6.3, our algorithm can also generate surface samples with a variety of blue noise properties, including distributions similar to CCVT. In Fig. 4.19 we compare our surface sampling results with Poisson disk sampling of [27] on a sphere. Our samples exhibit uniform distributions similar to the CCVT profile. In Fig. 4.20 we demonstrate a surface sampling on a bunny model and compare the Differential Domain Function (DDF) [130] with that of 2D CCVT [18]. The results look qualitatively similar – the difference is mainly due to the geodesic distance approximation when evaluating surface DDF.

Another interesting property of our algorithm is that samples implicitly distribute along ridges and valleys on the mesh. This is because in high curvature regions, points will
be automatically pushed outwards along the normal, attracting samples to regions of high curvature. This effect is actually beneficial for many sampling applications such as rendering and remeshing, capturing high frequency features on the mesh. Its application will be discussed in Section 4.6.4.

**Adaptive Sampling.** Adaptive sampling is achieved by applying a distance field scale $s(x)$ to the sample properties. As in previous work [130, 138], the distance field in 2D can be calculated from the intensity image $I(x)$ as $s(x) = 1/\sqrt{I(x)}$, or in 3D it can be defined as the inverse of the size function. The unwrapped distance between two points now becomes:

$$\tilde{s}(x_i, x_j) = \frac{2(x_i - x_j)}{s(x_i) + s(x_j)} \quad (4.47)$$

Which is then used in the kernel functions in Equation 4.37:

$$W_{ij} = W(\tilde{s}(x_i, x_j), h) \quad (4.48)$$

As we increase the scale, the kernels will cover larger regions. In equilibrium, to keep a similar density $\rho$ as defined in Equation 4.38, the total number of points (and their distri-
Figure 4.20: Surface sampling of a bunny model. Fig. 4.20(a) visualizes the samples. Fig. 4.20(b) shows the Differential Domain Function (DDF) of the bunny surface samples (averaged over 10 runs) and Fig. 4.20(c) shows the DDF of the CCVT method [18], both calculated using the method of Wei and Wang [130].

Figure 4.21: In a and b we show the result of Fattal [43] and our method respectively.
bution pattern) in the large kernel region shall remain similar to those with smaller scale. As a result, points will become more sparse in high scale regions and denser in low scale regions. The update of the sample position therefore needs to correspond to the distance field scale as well:

\[(x - x') \leftarrow \tilde{s}(x)(x - x').\]  \hspace{1cm} (4.49)

Notice that when computing the convergence criterion, one still needs to use the un-scaled distance instead of the new one.

<table>
<thead>
<tr>
<th>0.83%</th>
<th>9.05%</th>
<th>30.80%</th>
<th>59.32%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) Quadratic density ramp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0.9%</th>
<th>8.9%</th>
<th>30.9%</th>
<th>59.3%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) [18]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0.8%</th>
<th>10.2%</th>
<th>29.6%</th>
<th>59.4%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c) [43]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0.9%</th>
<th>9.2%</th>
<th>29.6%</th>
<th>60.3%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d) Our method</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.22: Comparison of adaptive sampling of our method vs. [18] and [43]. The spatial density function is defined as the quadratic ramp shown on the top. Every example contains 1000 points, the percentages underneath indicate the density ratios for each quarto of the ramp.

Image stippling is a common problem in Computer Graphics. In Fig. 4.21 we compare our method with the work of Fattal [43]. We additionally compare the quality of adaptive
sampling in Figure 4.22 with Balzer et al. [18] and Fattal [43] using a quadratic ramp as the scale function. In the coarse region, our method does not produce as good results as in the dense region. It is because the choice of $h$ limits the influence range. If $h$ is too large, it will cause the points to blur the given density function.

4.6.3 Varying Kernel Functions

The choice of SPH parameters and kernels directly affect the sampling patterns and distribution properties. In this section we evaluate these choices experimentally.

Kernel Size. In $n$-dimensional sampling, the average step distance between adjacent samples $d$ can be approximated by $d = \sqrt[2n]{V/N}$, where $V$ is the total sampling area/volume and $N$ is the total number of samples. Alternatively, $d$ can be precomputed precisely by running the algorithm. According to the Nyquist–Shannon theorem, the sufficient sampling rate is twice the bandwidth of a band–limited function, so the relationship between $h$ and $d$ can be written as $h \geq 2d$. Therefore, $h$ should not be too small.

However if the object has features which are small enough, our sampling method may fail if the smooth radius ($h$) is chosen too large. In Figure 4.23, we demonstrate the case of an object with an internal boundary. The smoothing radius must be chosen to be smaller than the minimum feature size, otherwise the sampling on the boundary will fail.

Trade-off between noise and aliasing. Heck and coworkers [52] characterized the blue noise sampling property by its effective Nyquist frequency $\nu_{\text{eff}}$ and oscillation $\Omega$. The effective Nyquist frequency measures the size of the empty low frequency region in its Fourier power spectrum. A blue noise sampling with high $\nu_{\text{eff}}$ can effectively reduce the low frequency noise. On the other hand, oscillation $\Omega$ measures the variation in the non-zero region of a blue noise spectrum. High oscillation would results in structured aliasing [52].
Figure 4.23: If the smooth radius is smaller than the feature size $b$, the interior boundary will be correctly sampled. Otherwise the sampling on the interior boundary will be incorrect $c$.

Ideally the perfect blue noise shall have high $\nu_{\text{eff}}$ with low $\Omega$. But unfortunately high effective Nyquist frequency always comes with high oscillation, thus users have to make the trade–off between those two properties, or in other words, the trade–off between noise and aliasing. While most of the existing blue noise sampling methods are implicitly making such trade–off, few discuss how to have direct control over it. In my SPH sampling method, we can control the tradeoff between $\nu_{\text{eff}} - \Omega$ by varying the ‘density difference’ parameter.

Such trade–off has to do with the convergence conditions of our algorithm. As I mentioned before, there is a trade–off between the two convergence criteria of the algorithm. When we set the rest density $\rho_0$ closer to the average particle density $\bar{\rho}$ upon convergence, the first convergence condition will be more strictly satisfied. In this case, as is shown by experiments (Figure 4.15) samples will give lower $\Omega$ while compromising $\nu_{\text{eff}}$.

When the difference between $\rho_0$ and $\rho_i$ is large, the algorithm will adhere more strictly to the second convergence condition and produce results with high $\nu_{\text{eff}}$, at the cost of high $\Omega$ (as in Figure 4.14. Notice that those samples with hexagon patterns actually give the theoretical upper bound of effective Nyquist frequency [42].

One can easily control the trade–off between $\nu_{\text{eff}}$ and $\Omega$ by tuning the difference between $\rho_0$ and $\rho_i$, exposing a single user controlled parameter for this purpose. A lower difference
refers to both low $\nu_{\text{eff}}$ and $\Omega$, and a high difference gives samples with high $\nu_{\text{eff}}$ and $\Omega$. The experimental results are shown in Table 4.1. By changing the density difference, we can see a smooth transition on the Fourier power spectrum.

![Graph](image)

Figure 4.24: Effective Nyquist frequency and Oscillation changes as we increase the ‘density difference’ parameter.

To directly see how the density difference affects $\nu_{\text{eff}}$ and $\Omega$, I also plot their relation in Figure 4.24: providing a guide to achieving preferred $\nu_{\text{eff}}$ and $\Omega$. Figure 4.25 plots the achievable $\nu_{\text{eff}}$ against the corresponding $\Omega$ alongside other sampling methods for comparison. This curve describes what are the $\nu_{\text{eff}} - \Omega$ pairs that a sampling method can achieve. The curve of SPH sampling passes through Lloyd’s relaxation and is also close to the CCVT method, and can also give sampling patterns in a continuous range between these two methods. Methods that are optimized for low oscillation with high effective Nyquist frequency, such as the step function and single–peak function [52] give a better trade-off than our method, although these methods are not directly tunable.

As discussed above, the difference between rest density $\rho_0$ and average particle density $\bar{\rho}$ significantly effects the sampling quality. To have direct control of the density difference, we can adaptively calculate the average density of all points $\bar{\rho}$ and reset the rest density $\rho_0$ so that it is lower than $\bar{\rho}$ by the predefined density difference. For accuracy, only points not near the surface count towards the average density since the density of surface points are incorrect.
Sampling with different kernels. Another interesting question is how the sampling patterns are affected by the choice of kernel. I show experimental results of different kernels in Table 4.2. In this experiment I replace the kernel gradient of the pressure force (which controls the mapping from the density difference to the force) with a box kernel, tent kernel, quadratic kernel and an off–center double-peak kernel. The results show that sampling patterns are directly influenced by the kernel choice. The tent kernel and quadratic kernel give results with a blue noise distribution, but the other kernels generate distributions that are more like red or green noise. Further experiments and theoretical analysis would be needed to understand the relationship between the kernel and the resulting distribution.

4.6.4 Applications

Remeshing. Our surface sampling algorithm enables us to achieve excellent surface sampling with accurate control of the sampling number, which can be used for re-meshing. A good remeshing algorithm should keep sharp features of the mesh while keeping sample...
Table 4.2: Sampling results with different kernels.

<table>
<thead>
<tr>
<th>Name</th>
<th>Kernel</th>
<th>Samples</th>
<th>Spectrum</th>
<th>Radial Mean &amp; Anisotropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Box</td>
<td><img src="image" alt="Box" /></td>
<td><img src="image" alt="Samples" /></td>
<td><img src="image" alt="Spectrum" /></td>
<td><img src="image" alt="Radial Mean &amp; Anisotropy" /></td>
</tr>
<tr>
<td>Tent</td>
<td><img src="image" alt="Tent" /></td>
<td><img src="image" alt="Samples" /></td>
<td><img src="image" alt="Spectrum" /></td>
<td><img src="image" alt="Radial Mean &amp; Anisotropy" /></td>
</tr>
<tr>
<td>Quadrat.</td>
<td><img src="image" alt="Quadrat" /></td>
<td><img src="image" alt="Samples" /></td>
<td><img src="image" alt="Spectrum" /></td>
<td><img src="image" alt="Radial Mean &amp; Anisotropy" /></td>
</tr>
<tr>
<td>Double-peak</td>
<td><img src="image" alt="Double-peak" /></td>
<td><img src="image" alt="Samples" /></td>
<td><img src="image" alt="Spectrum" /></td>
<td><img src="image" alt="Radial Mean &amp; Anisotropy" /></td>
</tr>
</tbody>
</table>

points uniformly distributed. Our algorithm can easily generate uniform samples due to our blue noise property. As discussed in Section 4.6.2 our algorithm will naturally place points in regions with sharp features.

In Figure 4.26 we sample a bowl surface and mesh the result using Ball Pivoting algorithm [23]. Our algorithm preserves the features of the bowl without the need for any additional techniques such as subsampling in regions of high curvature, thanks to the properties of SPH sampling described in Section 4.6.2. Further improvement of these results by adaptive sampling based on surface curvature is a promising research direction.

**Adaptive Volume Sampling.** Adaptive volume sampling according to different features can be very useful for a variety of applications, such as variational tetrahedralization and volume rendering. SPH sampling allows us to choose a different size function based on a
Figure 4.26: The surface of the bowl is remeshed with 5000 samples. With our method, points are implicitly attracted to regions of high curvature.

particular application. In Figure 4.27 we sample the interior and surface of the bunny by determining the feature size using the medial axis of the manifold similar with Adams et al. [4]. The scale function of the adaptive sampling is set according to the computed size function. Other size functions, such as that of Alliez et al. [11] are also applicable.
Figure 4.27: The bunny model is adaptively sampled based on a size function deduced using the medial axis [4], resulting in dense sampling of the ears, tail and feet, and sparse sampling of the body. The cutaway demonstrates that both the surface and internal volume is well adaptively sampled. The radius and color of particles is set according to the size function.
CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this dissertation, I have discussed several shape design and optimization methods for content creation in 3D printing. As 3D printing becomes more available to consumers, there are increasing demands for new content creation methods to help inexperienced users create customized designs for printing. The algorithms I have studied in this thesis are generally designed to be automatic and reduce dependency on expertise knowledge; they are aimed to help novice users to create printable content easily, thus it enables more people to enjoy fabricating customized 3D shapes.

Geometric Dissection Puzzles. In Chapter 2, I have presented a computational algorithm for creating geometric dissection puzzles. The user provides two input shapes of equal area, and the algorithm will dissect one shape into pieces which can then be rearranged into the other shape. As is shown by experimental results, my algorithm can easily create 2D and 3D dissection puzzles that are non-trivial to solve. The algorithm begins by discretizing the input shapes onto a uniform lattice (square or triangular), and the algorithm iteratively (through multiple levels of optimization) solves the dissection problem by generating the smallest number of clusters such that there is a one-to-one and congruent matching between the two sets of clusters. Specifically, in the \(i\)-th level the algorithm tries to achieve a dissection with \(i + 1\) pieces. This is done by optimizing an existing dissection with random seeding, forward/backward copy-pasting and random label switching. After the optimization, a set of potential solutions is outputted, which will serve as good starting points for the next level. The multi-level optimization stops when a complete dissection solution is found. I have also extended the algorithm to incorporate area-based weight, other lattice
grids, dissecting multiple figures, and finally dissecting 3D shapes. I believe the algorithm and extensions provide a convenient tool for users to design a variety of different geometric puzzles.

One major limitation of the algorithm is that due to discretization, many figures cannot be exactly represented using a discrete lattice grid. They have to be rasterized, resulting in approximate shapes. Therefore the method is not meant to substitute analytic approaches to many dissection problems. Nonetheless, for the purpose of creating puzzles, the approximate shapes to be sufficient in many cases. Furthermore, our results may provide insights and useful initial solutions for discovering an analytic dissection.

Another limitation is that the user is given little control over the algorithm (except for adjusting the area-based term), thus it’s difficult to constrain the solution to have certain desirable properties. One example is the symmetry of the pieces, which is often desirable from an aesthetic point of view. Currently we do not consider such properties but believe it is possible to enforce user-defined constraints by modifying the objective function and including new terms. However, given that a dissection problem can often have multiple solutions at the same level (such as the examples shown in Figure 2.11 and 2.12), it’s possible to account for these properties when selecting the final best solution. A better way is to include a symmetry-based term in the objective function to actively enforce such constraints.

All puzzles demonstrated in Chapter 2 can be easily fabricated using 3D printers. The fabricated puzzles presents an interesting application for educational and entertainment purposes. There are other practical applications of the algorithm. For example, the 3D extension of the algorithm may be used to solve manufacturing problems, such as decomposing a furniture into as few pieces as possible to fit in a specific packaging box. Another example is to design furniture that can transform between different shapes to provide multiple functions.
There are several research directions for future work. The first thing is to investigate how to design 3D pieces that can lock with each other, providing a stable physical structure. Second, user-specified constraints may also be incorporated into the design, for example, by allowing the user to specify certain parts of the input that must remain integral pieces to prevent them from splitting. Third, there would be an extension of how to create hinged dissection puzzles, which is partly explored in Chapter 3.

**Boxelization.** In Chapter 3, I have introduced a novel method for folding 3D shapes into cubes and boxes. The user provides an arbitrary input model, i.e., a car, a truck, or a bunny, and my algorithm will automatically split the model into connected (hinged) pieces, and find physically plausible folding paths, such that the pieces can be successfully folded into the destination box. Objects designed with the method can be physically printed and folded from one shape to the other without collisions. To do so, we segment the input shape into a set of voxels and find a tree that connects these voxels with joints. In details, the algorithm is divided into three major steps: voxelization, tree fitting, and interactive folding. In the voxelization step, we find a good segmentation of the input shape that reduces small pieces. Then, in the tree fitting step, beam search and simulated annealing is applied to find the joint types and locations that minimize our energy function. Collisions during folding is temporarily ignored in this step. Finally, in the interactive folding step, we use a physics simulator to unfold both the source and target shapes in order to validate that a collision-free folding sequence can be generated for the computed solution.

There are several directions for future work. Firstly, it is difficult to fabricate tight-fitting joints for 3D printing, and this causes our final output, which is printed in a single piece, to be weaker than desired. One potential solution to this problem is to modify the geometry of the joints [15, 28], but such techniques are designed for larger joints and are difficult to apply to our intricate results. Fortunately, digital manufacturing technologies are constantly improving, so we expect our fabricated results would become more stable in future. Also, because the printed joints are very lose, we resort to applying a small amount
of glue or a putty to hold parts together when folded. To produce more robust models, we would need to design hooks, pegs, or other types of retention system.

Then, the qualities of the computed solution and the fabricated result are not perfect. There is a trade-off between the amount of inner void and the completeness of the outer surface of the folded shape. This can be changed, if desired, by modifying the surface energy (3.8).

Also, our automatic voxelization method does not honor important features of the model, such as the eyes on the bunny and wheels on the car. A user interface to specify important features would be useful for such task. Note that because we divide our algorithm into three stages, it is possible to run voxelization with user-specified constraints in an interactive manner, without affecting the optimization stages.

Next, we only included hinge joint in our examples. Other joints, such as prismatic, cylindrical, or even linkages, would allow for more transforms. A telescoping joint would be very interesting to add, since this would enable us to hide a piece inside another larger piece.

In addition, we show that the template does not necessarily need to be a box. But because of our formulation, the target shape must be composed of voxels. It is possible to explore ways to carve away the inside of the shapes to make them transform to other shapes as well. Also, it would be useful to add another energy objective that allows the designer to specify where in the target shape each segmented voxel maps to. This could potentially allow the shape to transform more naturally to the target shape.

Our physics simulation result serves as a useful guide for folding and unfolding the shape. But it would work better if we could generate a step-by-step manual rather than a continuous animation.

Finally, it is possible to connect multiple outputs from our system to create one big output. For example, it would be amusing to create a robot where the head is made of a bunny, the torso from an elephant, the arms from kittens, and the legs from dragons.
**Design Optimization.** In Chapter 4 we introduce an algorithm to optimize 3D design for its functionality. Our algorithm can help novice users to make their designs more endurable and functional. The algorithm applies physics simulation to evaluate the stress distribution over the design, which then works with a shape dissimilarity metric to preserve the shape of important features during deformation. Then we reformulate the problem as a constrained optimization problem and solve it by a Newton-based optimizer. Experiments show that our algorithm can significantly reduce the stress distribution on the design by making subtle changes.

To better tetrahedralize the mesh, we also discuss a new sampling method based on Smoothed Particle Hydrodynamics. It is fast and provides controllable blue noise spectral profile. By adjusting the rest density we are able to generate a variety of blue noise sample patterns, with matching distribution properties that range from Lloyd’s relaxation and CCVT. The flexibility of SPH allows spatially–varying point density, leading to adaptive sampling. Our method supports multi–class blue noise sampling and surface/volume sampling in any dimension. Moreover, SPH is easy to implement in parallel which ensures the sampling efficiency. We have also performed experimental studies of the choice of SPH kernel and its influence on the resulting samples. Experiment results show that such method is promising in remeshing task.

There are a few directions for future work. Firstly, the current optimization method is limited to optimize stress distribution over an input mesh. It would be interesting to extend our algorithm to other domains which requires physics analysis, such as toy airfoil designing. Secondly, non-physics based attributes, such as the aesthetic of a design, would be also an interesting optimization goal. By replacing my physics simulator with other evaluation metric, our algorithm may be able to optimize for non-physics based attributes, which is a very interesting research direction.

In terms of the SPH-bases sampling method, there are also several directions for future research. The relationship between the kernel and its resulting sample spectrum is worth
further investigation. An interesting problem is to study how to compute, given a target spectral distribution function, a kernel function that would lead to that target distribution. This would make it possible to use my SPH-based method to generate samples with general spectral noise, including green, red, and other types of noise. Another possible direction is to study how to improve the trade-off between Nyquist Frequency and oscillation.
Here is a list of my publications during my MS/PhD program:


APPENDIX B

ANALYTIC GRADIENT AND HESSIAN OF STRAIN TENSOR $\epsilon$

Similar to Equation 4.25 and Equation 4.26, we can express the partial derivatives of the strain tensor $\frac{\partial \epsilon(i,j)}{\partial x_{v,k}}$ wrt the coordinates of a vertex in the optimized shape (here $r, s$ are matrix element indices):

$$\frac{\partial \epsilon}{\partial x_{v,k}} = U^T \bar{P} \frac{\partial \bar{X}^{-1}}{\partial x_{v,k}} V$$

$$\frac{\partial \bar{X}^{-1}(i,j)}{\partial \bar{X}(r,s)} = -\bar{X}^{-1}(i,r) \bar{X}^{-1}(s,j)$$

$$\frac{\partial \bar{X}(i,j)}{\partial x_{v,u}} = \begin{cases} 
1 & \text{if } v = v_i \text{ and } u = j \\
-1 & \text{if } v = v_0 \text{ and } u = j \\
0 & \text{otherwise}
\end{cases}$$

The Hessian of $\epsilon$ contains three parts: $\frac{\partial^2 \epsilon}{\partial \bar{P}^2}$, $\frac{\partial^2 \epsilon}{\partial P \partial \bar{X}}$ and $\frac{\partial^2 \epsilon}{\partial \bar{X}^2}$, expressed as follows (here $w$ refers to another vertex index, $l$ is a coordinate index):

$$\frac{\partial^2 \epsilon}{\partial \bar{P}^2} = 0$$

$$\frac{\partial^2 \epsilon}{\partial p_{v,k} \partial x_{w,l}} = U^T \frac{\partial \bar{P}}{\partial p_{v,k}} \frac{\partial (\bar{X}^{-1})}{\partial x_{w,l}} V$$

$$\frac{\partial^2 \epsilon}{\partial x_{v,k} \partial x_{w,l}} = U^T \bar{P} \frac{\partial^2 (X^{-1})}{\partial x_{v,k} \partial x_{w,l}} V$$

We also need to compute the second order derivatives of $X^{-1}$. Let us denote $X(i,*)$ as the $i^{th}$ row in matrix $X$, and $X(*,j)$ as the $j^{th}$ column. Then we will have:
\[
\frac{\partial^2 (\bar{X}^{-1})}{\partial \bar{X}(r, s) \partial \bar{X}(t, u)} = \bar{X}^{-1}(s, t) \cdot \bar{X}^{-1}(\ast, r) \cdot \bar{X}^{-1}(u, \ast) \\
+ \bar{X}^{-1}(t, s) \cdot \bar{X}^{-1}(\ast, u) \cdot \bar{X}^{-1}(r, \ast)
\]
The analytic Hessian of elastic energy $U$ gives the gradient of the force, which is actually the Jacobian of the force equilibrium constraints. It can be computed as follows:

\[
\frac{\partial^2 U(\bar{X}, \bar{P})}{\partial \bar{P}^2} = \nu \frac{\partial \epsilon}{\partial \bar{P}} \otimes E \frac{\partial \epsilon}{\partial \bar{P}}
\]

\[
\frac{\partial^2 U(\bar{X}, \bar{P})}{\partial \bar{X} \partial \bar{P}} = \nu \frac{\partial \epsilon}{\partial \bar{P}} \otimes E \frac{\partial \epsilon}{\partial \bar{X}} + \nu \epsilon \otimes E \frac{\partial^2 \epsilon}{\partial \bar{X} \partial \bar{P}} + \frac{\partial \nu}{\partial \bar{X}} \epsilon \otimes E \frac{\partial \epsilon}{\partial \bar{P}}
\]

where $\otimes$ refers to tensor multiplication, and the partial derivatives of the volume $\frac{\partial \nu}{\partial \bar{X}}$ can be computed as:

\[
\frac{\partial \nu}{\partial \bar{X}(i, j)} = \frac{1}{6} \left( \bar{X}(i+1, j+1) \ast \bar{X}(i+2, j+2) - \bar{X}(i+1, j+2) \ast \bar{X}(i+2, j+1) \right)
\]

Here the indices of the matrix should wrap around when $i+2 > 3$ or $j+2 > 3$. Notice that $\frac{\partial^2 U(\bar{X}, \bar{P})}{\partial^2 \bar{X}}$ is not needed when computing the Jacobian.
APPENDIX D

ANALYTIC GRADIENT AND HESSIAN OF VON MISES STRESS

If we rewrite $\sigma$ in vector form:

$$\sigma = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{23} \\ \sigma_{31} \end{bmatrix}$$  \hspace{1cm} (D.1)

One might easily notice that Equation 4.28 can be written in the form: Where $M$ is a constant matrix:

$$M = \begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 \end{bmatrix}$$  \hspace{1cm} (D.2)

Thus, Equation 4.28 can be further rewritten into the form:

$$\dot{\sigma}^2 = \frac{1}{2} \dot{\epsilon} \otimes \text{EME} \epsilon$$  \hspace{1cm} (D.3)
Which is very similar to Equation 4.21. Thus its analytic gradient also has a similar form:

\[
\frac{\partial \hat{\sigma}^2(\bar{X}, \bar{P})}{\partial \bar{P}} = (U^T \bar{P} \bar{X}^{-1} V - I) \otimes \text{EME} \frac{\partial \epsilon}{\partial \bar{P}}
\]

\[
\frac{\partial \hat{\sigma}^2(\bar{X}, \bar{P})}{\partial \bar{X}} = (U^T \bar{P} \bar{X}^{-1} V - I) \otimes \text{EME} \frac{\partial \epsilon}{\partial \bar{X}}
\]

The Hessian is expressed as follows:

\[
\frac{\partial^2 \hat{\sigma}(\bar{X}, \bar{P})}{\partial \bar{P}^2} = \frac{\partial \epsilon}{\partial \bar{P}} \otimes \text{EME} \frac{\partial \epsilon}{\partial \bar{P}}
\]

\[
\frac{\partial^2 \hat{\sigma}(\bar{X}, \bar{P})}{\partial \bar{X} \partial \bar{P}} = \frac{\partial \epsilon}{\partial \bar{P}} \otimes \text{EME} \frac{\partial \epsilon}{\partial \bar{X}} + \epsilon \otimes \text{EME} \frac{\partial^2 \epsilon}{\partial \bar{X} \partial \bar{P}}
\]

\[
\frac{\partial^2 \hat{\sigma}(\bar{X}, \bar{P})}{\partial \bar{X}^2} = \frac{\partial \epsilon}{\partial \bar{X}} \otimes \text{EME} \frac{\partial \epsilon}{\partial \bar{X}}
\]
BIBLIOGRAPHY


