

Sondre Husøy

Constrained Generation of Voronoi Meshes using Inscribed Sphere Distance

Graduate thesis in MTFYMA

Supervisor: Knut-Andreas Lie

Co-supervisor: August Johansson, Øystein Andersson Klemetsdal

June 2023

Sondre Husøy

Constrained Generation of Voronoi Meshes using Inscribed Sphere Distance

Graduate thesis in MTFYMA

Supervisor: Knut-Andreas Lie

Co-supervisor: August Johansson, Øystein Andersson Klemetsdal

June 2023

Norwegian University of Science and Technology

Faculty of Information Technology and Electrical Engineering

Department of Mathematical Sciences



Norwegian University of
Science and Technology

Abstract

Voronoi meshes, also known as PEBI (Perpendicular Bisector) grids, have garnered considerable attention due to their ability to simplify the generation of general polygonal/polyhedral meshes. Constrained Voronoi meshes, in particular, have emerged as a significant area of research, as they are able to incorporate geometric features within the resulting mesh. However, the presence of sharp intersections and narrow spaces poses challenges to constructing such meshes, often necessitating compromises to the mesh integrity.

This thesis aims to address these challenges by introducing new methods for creating constrained Voronoi meshes in three dimensions. We propose a novel distance function, the Inscribed Sphere Distance, and employ this to enhance the stability of the mesh generation. Throughout this research, we will explore and evaluate the advantages and disadvantages of Voronoi meshing as a whole, shedding light on its potential benefits and limitations.

By employing the newly developed methods and leveraging the Gmsh mesh generator, we have successfully implemented a Voronoi mesh generator that excels in capturing narrow spaces and sharp intersections. This enhanced capability allows for the creation of meshes that accurately represent complex geometries with intricate details, resulting in improved simulation accuracy and fidelity.

Sammendrag

Voronoi mesh, også kjent som PEBI (Perpendicular Bisector) grids, har nylig fanget oppmerksomhet ettersom de forenkler konstruksjonen av polygonale/polyhedrale mesh. Betingede Voronoi mesh har spesifikt blitt et tema for forskning, ettersom de kan inkorporere geometriske trekk for å representere fysiske egenskaper i det ferdige meshet. Skarpe kanter og trange passasjer er derimot utfordrende å representere med denne typen mesh, og kompromisser til de geometriske trekkene må ofte gjøres for å opprettholde mesh kvalitet.

I denne avhandlingen takler vi denne problemstillingen gjennom introduksjonen av nye konstruksjonsmetoder for betingede Voronoi mesh. Med bruk av vår nye avstandsfunksjon kalt ISD (Inscribed Sphere Distance) sikter vi på å forbedre stabiliteten til mesh konstruksjonen. Fremgangsmåten vil også hjelpe oss kaste lys på fordelene og begrensningene til betingede Voronoi mesh i sin helhet.

Ved å utnytte disse nye metodene i samsvar med Gmsh sin mesh generator har vi utviklet en algoritme for Voronoi mesh konstruksjon som utpreger seg på skarpe kanter og trange passasjer. Disse egenskapene tillater konstruksjonen av mesh som presist inkorporerer kompleks geometri, hvilket vil resultere i bedre presisjon for simulering.

Preface

First and foremost, I would like to acknowledge my coffee maker, which, with unwavering dedication, replaced numerous potential hours of blissful sleep with countless hours of arduous work. I extend my deepest appreciation to my esteemed advisors, Knut-Andreas Lie, August Johanson, and Øystein Andersson Klemetsdal. Their exceptional guidance, unwavering support, and infectious enthusiasm have been instrumental in shaping this thesis.

Table of Contents

List of Figures	vi
1 Introduction	1
2 Theory	5
2.1 Voronoi	5
2.1.1 Voronoi metrics	5
2.1.2 Delaunay dual	6
2.1.3 Centroidal Voronoi meshes	8
2.2 Constrained Voronoi meshes	9
2.3 Cell constraints	11
3 Method	12
3.1 Subdividing face constraint	12
3.1.1 Polygonal constraint borders	12
3.2 Designating vertex sphere radius	13
3.3 Making centroidal cells	13
4 Development	15
4.1 Baseline implementation	15
4.2 Causes of stray sites	19
4.3 The inscribed sphere distance	19
4.3.1 Surface to triangle ISD	21
4.3.2 ISD cell proportion	24
4.3.3 Preliminary ISD test	26
4.4 Sharp edges	27
4.5 Minimum size padding	27
4.6 Fan-triangle padding	29
4.7 Line ISD	33

4.8	Point ISD	34
5	Implementation	35
5.1	ConstrainedEdgeCollection	35
5.2	TriangulatedSurface	37
5.3	Background sites	39
6	Gmsh Implementation	40
6.1	Basics	40
6.2	Mesh size	42
6.3	External process mesh-field	43
6.3.1	Meshing algorithms	44
6.4	Embedding	44
6.5	Transfinite curves	45
7	Results and Further Development	46
7.1	Misshapen fan edges	46
7.1.1	Curved constraints	49
7.1.2	Performance	50
7.2	Improvements to ISD	50
	Bibliography	52

List of Figures

1.1	Triangular mesh in \mathbb{R}^2 with different mesh sizes. The algorithm (Gmsh) produces areas with similar triangular patterns. The artifacting at the transition between these pattern is visible, despite increasing the mesh refinement.	2
1.2	A Voronoi mesh in \mathbb{R}^2 using Euclidean distance. It is constructed of randomly placed sites, shown in blue. The ridges, shown as black lines, are equidistant from a pair of sites. The vertices, shown in orange, are at the intersection of at least three ridges. The dashed lines are unbound ridges that intersect another ridge at one end only. .	2
1.3	Using the vertices of a triangle mesh generator (Gmsh in this case) as sites results in well formed Voronoi cells.	3
2.1	* A set of 5 sites, shown as grey dots, placed equidistantly from a target vertex, shown as a blue cross. The resulting Voronoi mesh has a vertex at the target position with 5 connected ridges, shown as blue lines.	6
2.2	Voronoi diagrams using different distance functions. The left side shows the distance to the closest sites as a heat map. The right side shows the resulting cells as different colors.	7
2.3	By mirroring the site of any cell that crosses a boundary(red), the resulting set of ridges between the original (blue) and mirrored (orange) sites will follow the border.	8
2.4	* Construction of a Voronoi mesh using a Delaunay triangulation of the sites. The circumcenters of the Delaunay triangulations become the vertices of the Voronoi mesh	8
2.5	Weirdly shaped centroidal Voronoi meshes.	9
2.6	Face constraint construction	10
2.7	* Face constraint breaking. On the left the face constraint is satisfied. On the right the face constraint breaks as a result of nearby unconstrained sites (yellow) breaking the second face condition by entering the vertex spheres (blue) of the constrained sites (grey). The resulting constrained ridges are no longer directly connected. . .	11
2.8	* An intersection between a face constraint and a cell constraint, handled by prioritizing one of the two constraints. The sites constrained by the face constraint (red line) are shown as grey points. The sites constrained by the cell constraint (blue line) are shown as yellow points.	11
3.1	The site height shown as a red line is determined in relation to the ridge edge to ensure that it maintains an equal distance from the mirrored site and neighboring sites on the same side of the constraint. The resulting vertex radius is depicted in blue.	14
4.1	Unit plane triangulated using Gmsh.	16

4.2	The sites corresponding to a given triangle, highlighted in blue, are placed at the intersection of the three spheres centered at the triangle’s vertices. Each vertex has only one sphere radius, used in the intersections of all the neighbouring sites. . . .	16
4.3	The minimum radii of the vertices of a triangle seen from above. The blue circles represent the intersection between the vertex spheres and the triangle plane. The red circles show the minimum radius of each vertex sphere. If any of the sphere radii are shrunk beyond these minimum values, the spheres no longer intersect at one point.	17
4.4	The maximum radii of a vertex seen from above. The blue circles represent the intersection between the vertex spheres and the triangle plane. The red arrow illustrates the maximum radius of one of the vertices. If this radius is exceeded, the sphere will encompass the entire intersection circle formed by the other spheres, instead of intersecting it.	17
4.5	Inscribed sphere distance for a given point, shown in red. The direction from the point to the sphere center is the normal of the plane at the position of the point. .	20
4.6	Inscribed distance being limited by the curvature of the surface	20
4.7	Plane-to-plane ISD: Inscribed distance between a point on a plane and a triangle when the sphere tangents the interior of the triangle. The blue and red lines are the normals of the plane and triangle, respectively	22
4.8	In the interior of the boundary, shown in green, the points on the plane will have inscribed spheres that tangents the interior of the constraint triangle.	22
4.9	Plane-to-line ISD: between a point on a plane and a line segment.	23
4.10	The point of contact between the inscribed circle and the two potential triangle lines is outside the ends of the line segment. Consequently the point of contact is on one of the triangle corners.	25
4.11	The ISD of a plane is evaluated by considering a single constraint triangle positioned above it. The heatmap visualization illustrates the continuity of the function, while the contour plot showcases the continuous gradient.	26
4.12	Test case for ISD-based density, consisting of two planes that gradually converge. .	28
4.13	Triangulation of the test case using ISD-based mesh density.	28
4.14	Triangulation of the test case with site locations inside(purple) and outside (yellow) the boundary	28
4.15	Voronoi mesh generated from the test case site locations.	28
4.16	When using a constant mesh size, the gradual narrowing between two intersecting planes result in highly contested space. Here the vertex spheres of one constraint, shown in blue, have the smallest radius possible for the constant spacing. The possible spaces the remaining sites can be placed is highlighted in green. Despite the small radii, the available space is very restricted.	29
4.17	Vertex spheres of a main-loop vertex, shown in red. The fan-like construction of the triangles allows the border vertices, shown in blue, to have matching sphere radius to the interior vertices, shown in green. This allows for the sudden jump from an arbitrarily high mesh density to one of significantly lower mesh density.	30
4.18	Fan triangle padding for a 30° intersection using different densities for the intersection vertices. The ISD across the plane is shown as a heat map(left)	31
4.19	Fan triangle padding for intersections at different angles, using the a vertex density of 0.1 at the intersection. The ISD across the plane is shown as a heat map(left) .	32

7.1	Voronoi mesh of a unit cube with one offset edge, causing a gradual sharpening of angles	47
7.2	Lightning bolt	48
7.3	Mesh containing sharp edges generated by the Vorocrust algorithm [1]. Image copied from https://vorocrust.sandia.gov , 09/06/2023.	48
7.4	The act of clipping a sharp intersection on a constraint surface yields fan-triangles that are better suited to complement the surrounding triangles.	49
7.5	A small constraint segment(blue) placed inside the vertex sphere(solid red). The ISD(dotted red) at a nearby location is large as a result of its unusual placement and small size	51

Chapter 1

Introduction

Discretization of volumes into meshes is an important step in many numerical methods and simulation, such as finite element analysis [19] and finite volume methods [16]. Numerical simulations require boundary shapes to be divided into cells. Higher refinement of these cells improves the numerical approximations, but often at a significantly increased computational cost. While the computational capabilities of modern computers push the boundaries of numerical simulation, we wish for meshing methods to improve the quality of numerical methods without needlessly increasing the refinement of the mesh.

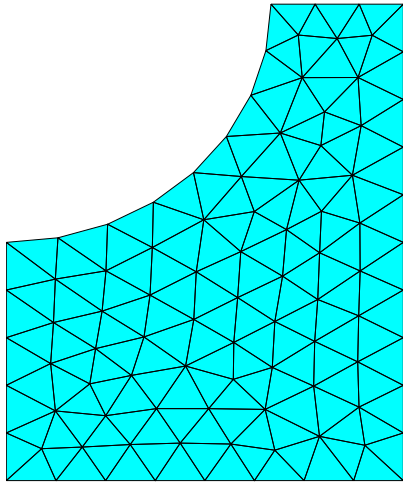
Unstructured meshes such as tetrahedral meshes allow for varying cell density across the given space. This is utilized in methods such as adaptive mesh refinement, where the cell density is increased in areas where the simulated properties or input parameters have large variations. Unstructured meshes also allow us to align the faces of the cells with internal and external boundaries of the simulated object that needs to be represented accurately. One example of such boundaries are fractures and wells in the simulation of flow in porous media[15, 13].

The typical cell shape for unstructured meshes are simplices such as triangles or tetrahedra. While these meshes are highly flexible and allow for alignment with complicated shapes, they often produce artifacting when aligned with multiple features simultaneously, see Figure 1.1. Aligning the cells to a given face causes the triangles to organize in a structured pattern extruding from the faces. At the transition between such patterns, the mesh will contain visible artifacting in the form of triangles that do not match either of the patterns.

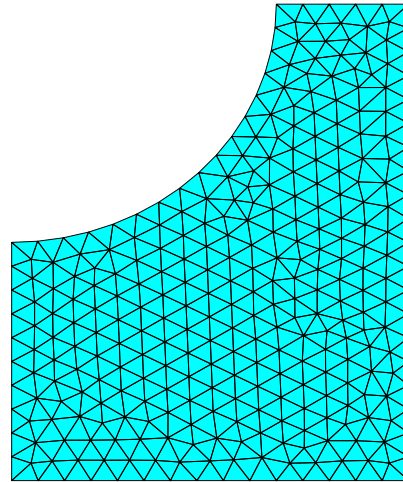
Meshes constructed from general polygonal cells have recently become a subject of interest to solve this issue. The more flexible cell shapes allow for meshes of lower refinement to accurately align its faces with any number of features simultaneously. Simpler variants like Quad based cells for 2D and 3D meshes have already become widespread, and further development into unstructured polygonal meshes of varying cell shapes has also shown promise[4].

The problem is that construction of such meshes is a far more unexplored field than traditional simplex meshes. Robust algorithms for triangular meshes such as Delaunay refinement already exist. These methods place the cell vertices based on clear criteria for the internal angles and size of each cell. Such criteria are significantly more difficult to define when the cells can have an arbitrary number of vertices. Additionally, aligning mesh faces with features is a more restrictive problem for polygonal cells, since the cell faces still need to be planar. This often makes it difficult or impossible to move a single vertex without adjusting the vertices of all its adjacent faces. While the benefits of polygonal/polyhedral meshes are clear, the construction of polygonal/polyhedral cells through vertex placement is a hurdle that so far has prevented them from replacing traditional simplex meshes.

A method that avoids this problem entirely is the generation of Voronoi meshes, see Figure 1.2. These meshes are constructed from a set of points referred to as sites or seeds. Each site corresponds to a cell, meaning the cell density is determined by the density of the sites. The vertices themselves are never directly placed, but rather determined by the placement of these sites. Unlike the planar



(a) Mesh size = 0.1



(b) Mesh size = 0.05

Figure 1.1: Triangular mesh in \mathbb{R}^2 with different mesh sizes. The algorithm (Gmsh) produces areas with similar triangular patterns. The artifacting at the transition between these pattern is visible, despite increasing the mesh refinement.

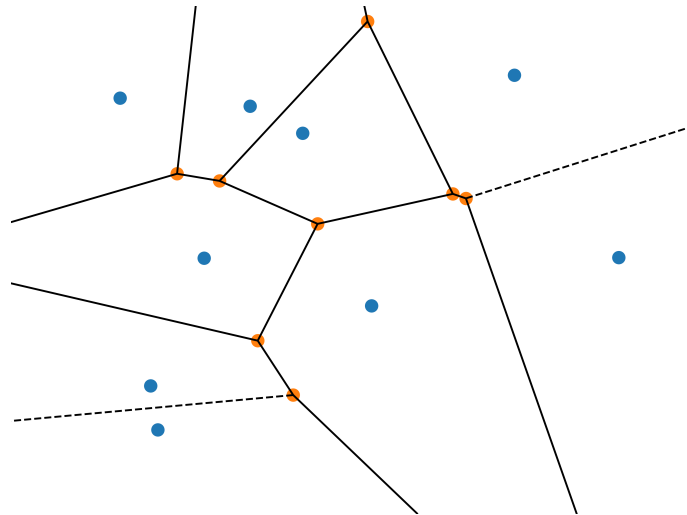


Figure 1.2: A Voronoi mesh in \mathbb{R}^2 using Euclidean distance. It is constructed of randomly placed sites, shown in blue. The ridges, shown as black lines, are equidistant from a pair of sites. The vertices, shown in orange, are at the intersection of at least three ridges. The dashed lines are unbound ridges that intersect another ridge at one end only.

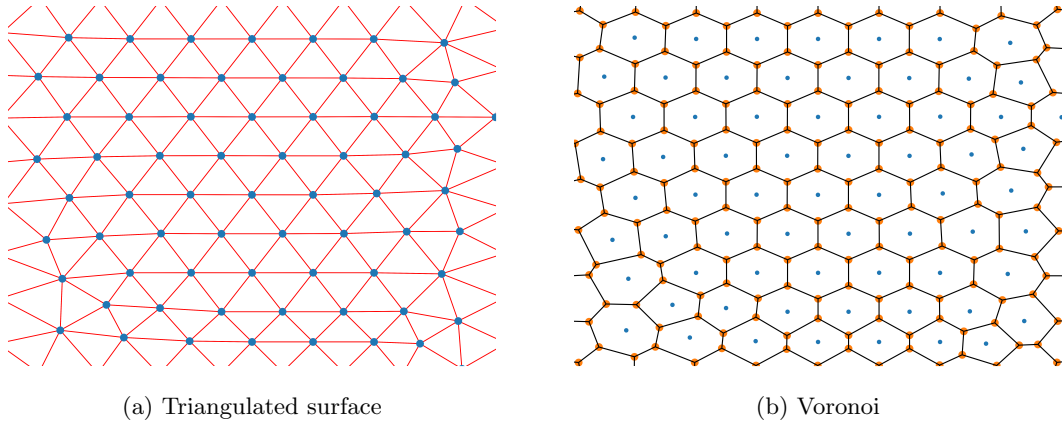


Figure 1.3: Using the vertices of a triangle mesh generator (Gmsh in this case) as sites results in well formed Voronoi cells.

restrictions of vertices in polygonal faces, these sites are entirely unrestricted, making them easy to distribute. The generation of Voronoi meshes from a set of site locations is also a well known problem with stable, performant solutions [3, 21].

Additionally, simple criteria exist to evaluate the quality of the mesh cells [14]. One such criterion is how centered the sites are in their respective cells. Traditional triangular meshing algorithms have been shown to produce high quality site locations, where the resulting cells are approximately centroidal, see Figure 1.3.

By relinquishing direct control of the mesh vertices, Voronoi meshes greatly simplify the construction of polygonal cells, allowing for high quality polygonal meshes with great flexibility in cell shape and adaptive refinement.

While there are clear benefits to this approach, there are still obstacles to overcome. Aligning mesh faces to a set lower-dimensional geometric objects requires restricted positioning of the mesh vertices. The lack of direct control over the vertex position forces us to resort to indirectly positioning the vertices through constrained positioning of neighbouring sites. This is referred to as constrained Voronoi mesh generation. A set of constricted sites is generated that results in a mesh with faces aligned with the desired geometric objects.

The generation of such constricted sites is no simple task. The relation between the constricted sites and the vertices they aim to enforce is entirely based on proximity. This makes it easy for constricted sites to interfere with one another. In order to construct a constrained Voronoi mesh this interference must be prevented. The general approach to this is to determine the position of constricted sites using the intersection of spheres placed at the target vertex positions. Such an algorithm needs to find a suitable set of vertex position with sphere radii that satisfy a set of conditions.

The UPR package, developed by Berge et al. [4, 18, 5], employs an approach that involves generating Voronoi meshes while adhering to a set of constraints that define specific characteristics of the mesh. This package is a component of the Matlab Reservoir Simulation Toolkit (MRST), which is a comprehensive Matlab toolkit designed for the simulation of geological reservoirs. The constraints integrated into the UPR package pertain to geological elements, including fractures and sedimentary layers. However, when numerous constraints are located in close proximity and feature acute intersections, the algorithm encounters difficulties in generating an appropriate mesh. In such scenarios, the algorithm deviates from exact constraint conformity and instead prioritizes the attainment of a reasonable mesh quality.

Vorocrust, a Voronoi mesh generator [1], uses a different approach to guarantee a valid Voronoi mesh. The algorithm initially attempts to generate the mesh with a predetermined refinement level. However, if this process fails, it recursively incorporates finer geometry until a conforming Voronoi mesh is successfully generated.

It is important to note that neither of these methods can preemptively determine the resolution and discretization of the final mesh. The UPR method compromises on the conformity to the imposed constraints, while Vorocrust employs a trial-and-error approach to refine the geometry until the desired Voronoi mesh is achieved.

In this thesis, we will develop new methods to create constrained Voronoi meshes in 3D space (\mathbb{R}^3). Our approach involves using a new distance function called the inscribed sphere distance (ISD). By using ISD as a criterion for mesh density, we can estimate how finely we need to refine the mesh to ensure it meets the given constraints. This approach will also help us understand the pros and cons of Voronoi meshing by giving us a clear indication of what makes certain scenarios challenging to mesh.

Chapter 2

Theory

This chapter will go over the theory behind Voronoi meshes, as well as previous work done by Berge et al. [4, 18, 5] and the authors specialization assignment [10]. While the chapter has been rewritten, it will bear significant resemblance with the work done in these works. Additionally, some figures will be reused from the specialization assignment. These figures will be marked with a blue asterisk(*) for clarity.

2.1 Voronoi

A Voronoi grid, also known as a PEBI¹ grid, is a grid constructed from a given metric space Ω and a set of points $s_i \in S$, herein referred to as sites; see Figure 1.2. Each cell c_i of these grids corresponds to one of these site-locations and is described by the following definition:

Definition 1. *Voronoi cell: For a given site $s_i \in S$, the corresponding cell c_i is the space around s_i where no other site in S is closer, defined by the expression*

$$c_i := \{x : x \in \Omega, \|x - s_i\| < \|x - s_j\|, \forall s_j \in S \setminus s_i\}.$$

A cell can therefore also be defined as the interior of all the ridges its corresponding site forms.

Definition 2. *Voronoi ridge: A ridge r_{ij} between two cells, c_i and c_j is the border between the two cells where the distance to their corresponding sites s_i and s_j is equal.*

$$r_{ij} := \{x : x \in \Omega, \|x - s_i\| = \|x - s_j\| < \|x - s_k\|, \forall s_k \in S \setminus \{s_i, s_j\}\}.$$

It is worth noting that not all pairs of sites share a non-empty ridge. Sites and cells that do share a non-empty ridge are called neighbours.

Voronoi vertices are formed at the intersection of at least $n + 1$ ridges, where n is the number of dimensions in the space. These are points where the $n + 1$ closest sites are equidistant. It is possible for more than $n + 1$ sites to be equidistant to a vertex, resulting in vertices with more than $n + 1$ ridges connected, see Figure 2.1.

This highlights the most useful and simultaneously troublesome feature of Voronoi meshes; we do not directly decide vertex placement. To move a vertex or ridge, we must move one or more of its sites.

2.1.1 Voronoi metrics

Voronoi grids can be constructed in any metric space; see Figure 2.2. The most commonly used metric is the Euclidean distance. In general, one may assume that this metric is used for Voronoi

¹Perpendicular Bisector

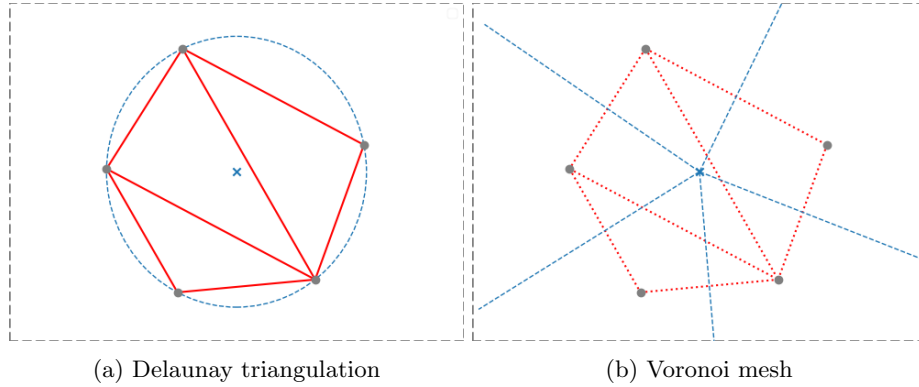


Figure 2.1: * A set of 5 sites, shown as grey dots, placed equidistantly from a target vertex, shown as a blue cross. The resulting Voronoi mesh has a vertex at the target position with 5 connected ridges, shown as blue lines.

meshes, unless specified otherwise. There are several reasons for this, the most notable being how it simplifies the ridges.

In metrics such as the Chebyshev distance or rectilinear distance, the ridges between two cells form jagged borders, see Figure 2.2. In Euclidean distance for \mathbb{R}^2 and \mathbb{R}^3 , the ridges of the grid are lines or planes. This leads to the final mesh consisting of the interior of its ridge, which in turn can be represented as the convex hull of its vertices.

One exception to this is unbound cells typically formed by the outermost sites. These cells are not closed and are defined as the interior of unbound ridges stretching out infinitely. However, most applications are only concerned with the interior of the given domain. Because of this, the unbound cells are typically removed from the final mesh or clipped to fit the boundary. A noteworthy feature of clipped Voronoi meshes is how they remain valid Voronoi meshes as long as the boundary can be represented by a convex polyhedron. This is easily shown, as the clipping can be mimicked by mirroring the corresponding sites to any cell that crosses the boundary; see Figure 2.3.

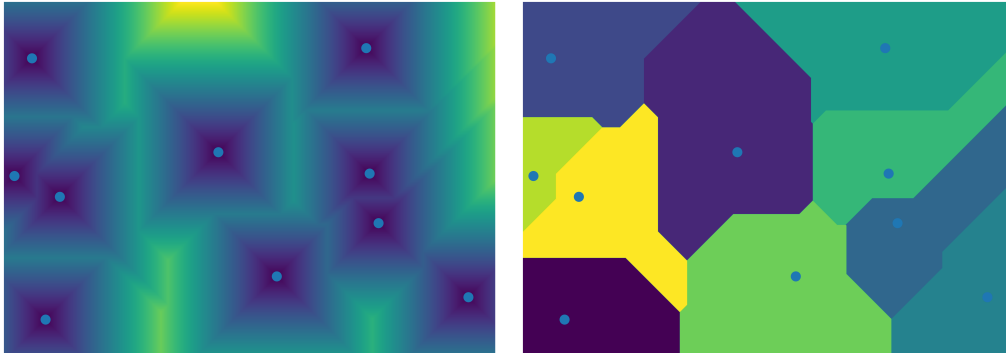
In this thesis this will not be necessary, the construction of the boundary will utilize the same approach as the constraints, resulting in a mesh perfectly contained by its boundary once we remove the cells not within it.

2.1.2 Delaunay dual

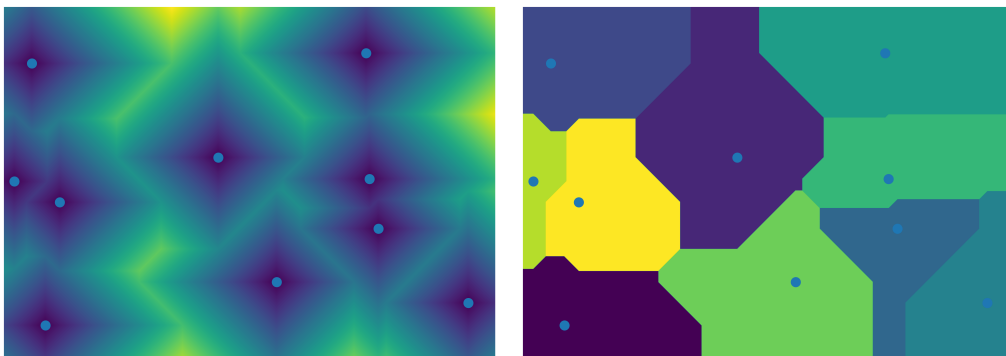
The generation of Voronoi meshes from a given set of sites is a well known problem with robust solutions. In this thesis we will utilize the Voronoi class of the Scipy [20] library. This library uses the QHull algorithm [3] to generate Voronoi meshes. Algorithms such as QHull utilize the fact that Voronoi meshes are the dual of the Delaunay triangulation of the given sites [6]; see Figure 2.4.

A Delaunay triangulation of a set of points is simply the triangulation that maximizes the smallest angle in any of the triangles. It has the property that no point of any triangle/tetrahedron is inside the circumscribed circle/sphere of any other triangle/tetrahedron. At least one such triangulation exist for any set of points, unless all the sites are on the same hyperplane.

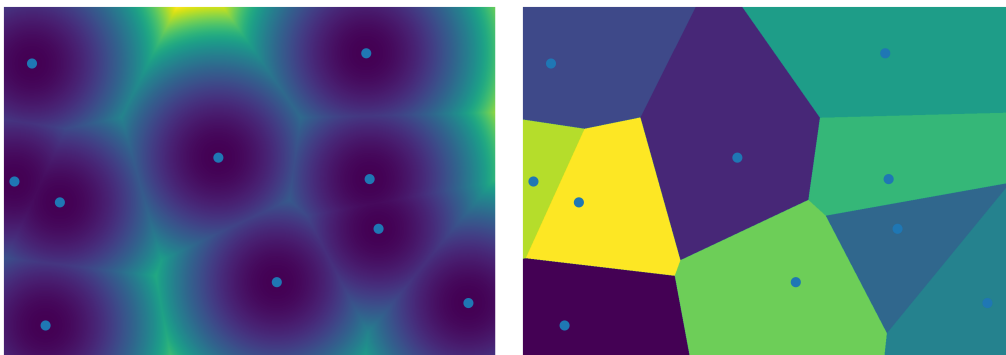
Every pair of sites sharing an edge in the Delaunay triangulation will share a ridge in the Voronoi mesh; see Figure 2.4. In the case where multiple valid Delaunay triangulations exist, they will all yield identical Voronoi duals. This happens when more than $n + 1$ neighbouring sites are equidistant from a given point. This results in all of the corresponding cells sharing a single vertex, as previously shown in Figure 2.1. This is noteworthy, since constrained Voronoi site generation will often place multiple sites equidistantly from a given point, resulting in such vertices.



(a) Maximum distance



(b) Chebyshev distance



(c) Euclidean distance

Figure 2.2: Voronoi diagrams using different distance functions. The left side shows the distance to the closest sites as a heat map. The right side shows the resulting cells as different colors.

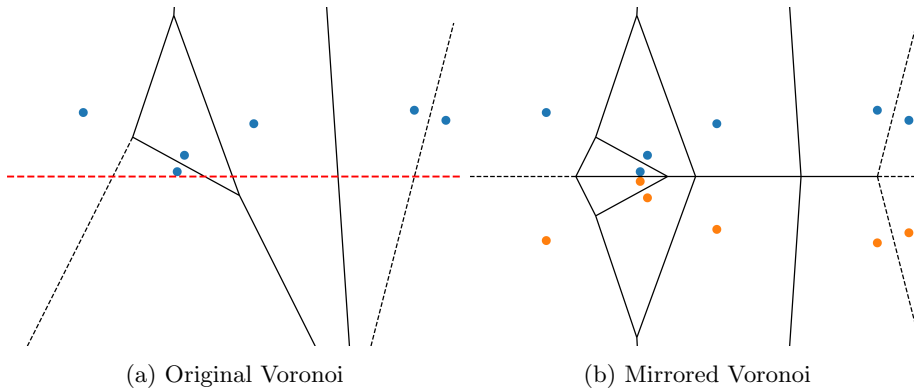


Figure 2.3: By mirroring the site of any cell that crosses a boundary (red), the resulting set of ridges between the original (blue) and mirrored (orange) sites will follow the border.

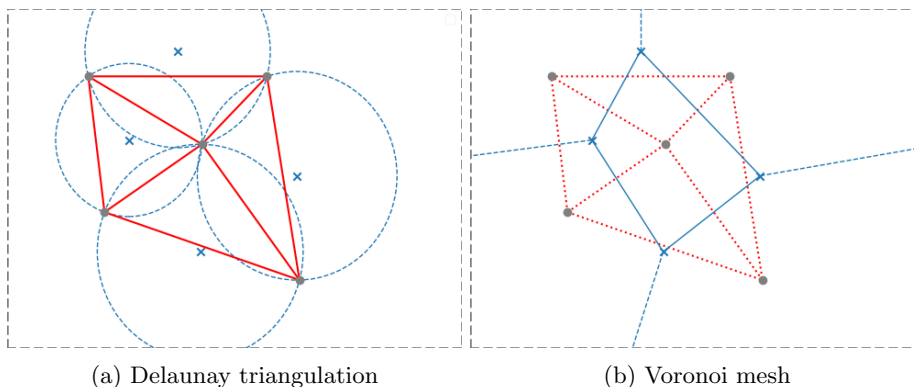


Figure 2.4: * Construction of a Voronoi mesh using a Delaunay triangulation of the sites. The circumcenters of the Delaunay triangulations become the vertices of the Voronoi mesh

2.1.3 Centroidal Voronoi meshes

A benefit of Voronoi meshes is the availability of criteria for mesh quality. One such criteria is the CPG energy function [12, 14]. This function is defined as the mean squared distance from a site to its cell interior

$$F(S) = \sum_i^n \int_{\vec{x} \in c_i} \|\vec{x} - \vec{s}_i\|^2, \quad (2.1)$$

We may also add a density function $\rho(\vec{x})$ to the equation. This generalizes the equation for meshes where the desired cell density varies

$$F(S) = \sum_i^n \int_{\vec{x} \in c_i} \rho(\vec{x}) \|\vec{x} - \vec{s}_i\|^2. \quad (2.2)$$

The energy function reaches its minimum when the Voronoi cells are centroidal, meaning that the site locations overlap with their corresponding center of mass. Centroidal Voronoi meshes have been shown to have numerous applications within fields like numerical simulation and compression [14, 7].

Note that it is possible to construct centroidal Voronoi meshes that are not minima of the energy function, see Figure 2.5.

If we consider the energy function and the criteria for triangular meshes, we notice an interesting property. Triangular meshing algorithms typically try to maintain equal distance between neighbouring vertices [17]. This is done to keep neighbouring triangles similar in shape and area. If

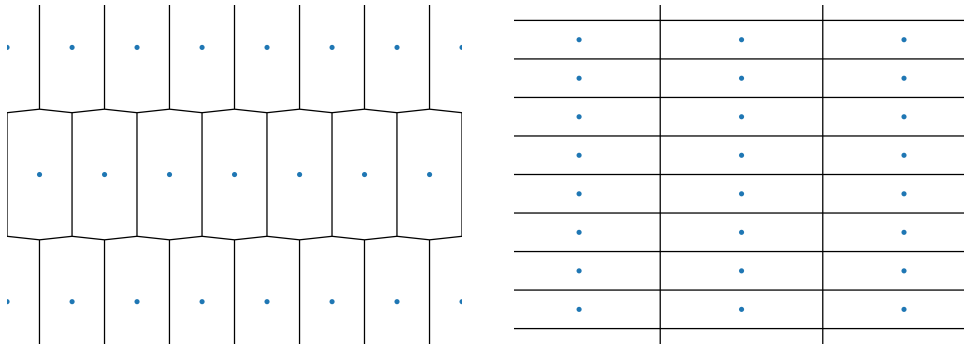


Figure 2.5: Weiridly shaped centroidal Voronoi meshes.

a Voronoi site is surrounded by equidistant sites, the corresponding cell becomes centroidal. Because of this, many triangular meshing algorithms are well suited to place Voronoi sites as well, see Figure 1.3. Examples of this has been shown by Berge et al. [18, 5, 4], and the authors project assignment [10], where conventional triangular and tetrahedral mesh generators have been utilized to fill in the sites of the Voronoi mesh.

2.2 Constrained Voronoi meshes

We wish for faces of the final mesh to be aligned with internal and external boundaries (or other lower-dimensional objects). When using more traditional unstructured meshes, such as triangular meshes, the vertices can be modified directly in order to align the faces.

For Voronoi meshes, however, this is not as easy. The vertices of the final mesh is not directly specified, but decided through the position of the closest sites. If we were to move some of the vertices of the final mesh, the resulting mesh would no longer be a Voronoi mesh. While straying from Voronoi meshing is a possible solution, the act of moving the vertices in the first place is difficult if we want to maintain polygonal faces in the cells. The difficulty of this problem is part of the reason why Voronoi meshes are beneficial to begin with.

In order to construct a mesh that conforms to the constraints, we look into creating a set of constrained site locations that result in ridges covering the desired surfaces. Following the notation of the UPR module [4, 18, 5], we introduce the concept of a face constraint. A face constraint describes a line or surface, depending on the dimensions of the mesh, that we wish to reproduce as accurately as possible by a set of connected ridges. This means that all the vertices of the constrained ridges must lie on the corresponding line or surface of the face constraint. We also require these ridges to form a connected surface. The method of generating these sites will now be described, a visual representation in \mathbb{R}^2 is shown in Figure 2.6.

Voronoi vertices form at positions where at least $n + 1$ sites are equidistant and no other sites are closer, n being the dimension of the space. Because of this, we can ensure that a given vertex position is included in the final Voronoi mesh by ensuring that all the $n + 1$ closest sites are the same distance from the vertex. In principle, this means that all of these sites need to lie on some sphere or circle centered on the vertex position; see Figure 2.1. We refer to these spheres and circles as vertex spheres. We introduce the following two definitions:

Definition 3. *Target vertex: The desired position of a vertex in the final mesh.*

Definition 4. *Vertex sphere: A circle or sphere centered on a target vertex to assist with the positioning of constrained sites.*

The approach therefore starts with designating a set of target vertices, each with a corresponding sphere radius. We require the constrained sites to be placed on the surface of these vertex spheres.

Additionally, we need the vertices to be connected by ridges. For a ridge to connect a set of vertices, all the vertices need to share some site. These sites therefore need to be on the vertex sphere of

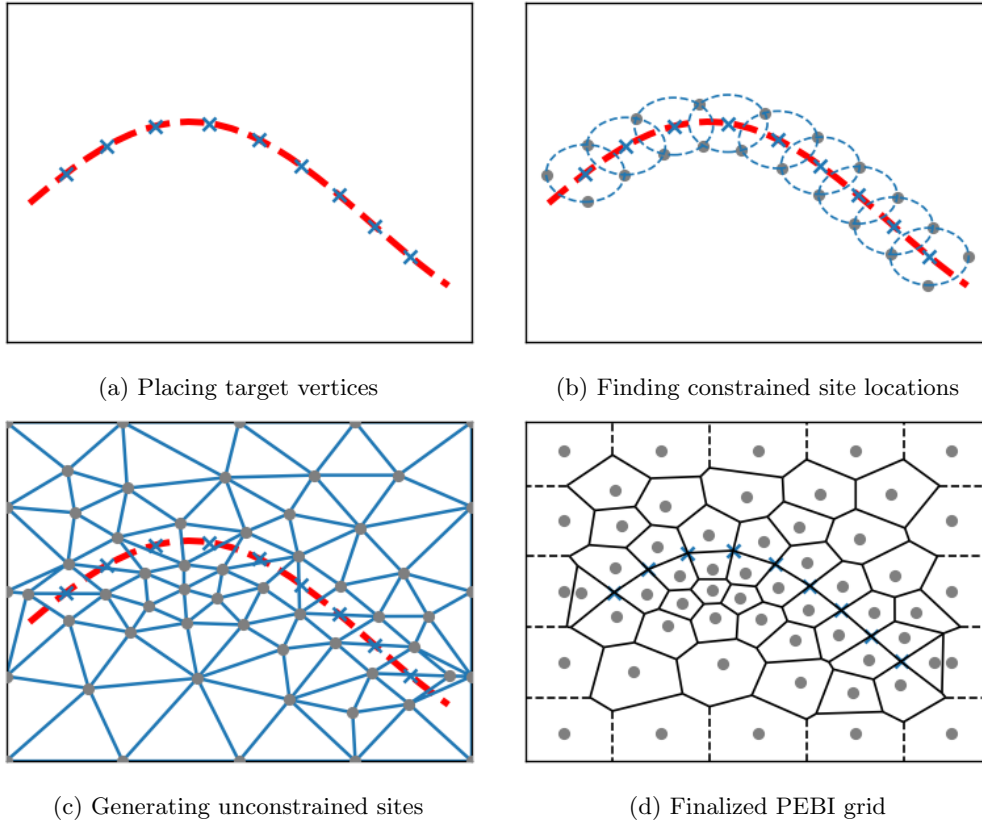


Figure 2.6: * The construction of a face constraint. **a)** The target vertices, shown as blue crosses, are placed evenly along the given line. **b)** The constrained sites, shown as grey dots, are found at the intersection of circles centered at the vertices. **c)** Using an FEM grid generator, the remaining sites are distributed. **d)** The PEBI grid generated using the sites has ridges following the constraint line

all the connected vertices. As such we place the sites on the intersections of the connected vertex spheres.

For this method to work, two conditions must hold, as described by Berge et al. [4, 18, 5]: First, the vertex spheres of a target ridge needs to intersect at one point on either side of the ridge. This is necessary to get the position of the two sites the ridge belongs to.

Definition 5. *Face condition 1: For any given target ridge, its vertex spheres must all intersect at two points.*

The second condition is that no site should be in the interior of any vertex sphere. Since the vertex placement is entirely based on the position of its closest sites, introducing a site in the interior of any vertex sphere will result in the vertex being moved and the ridge connections being broken; see Figure 2.7.

Definition 6. *Face condition 2: No site can be in the interior of any vertex sphere.*

We refer to a site breaking this condition as a stray site.

Definition 7. *Stray site: A site placed in the interior of a non-neighbouring vertex sphere.*

As such, the number of vertices in each target ridge must be equal to the dimensionality of the space. For \mathbb{R}^2 , the ridges are one-dimensional line segments, meaning this restriction does not change much. For \mathbb{R}^3 , however, the restriction means the face constraint planes are divided into triangles. This means that while a Voronoi mesh in \mathbb{R}^3 consists of polygonal cells, the ridges bordering the constraints in meshes constructed with this method will always be triangular.

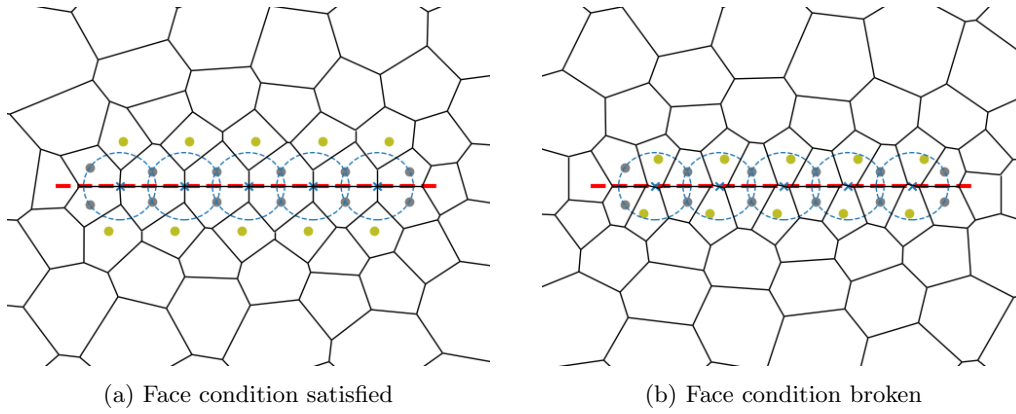


Figure 2.7: * Face constraint breaking. On the left the face constraint is satisfied. On the right the face constraint breaks as a result of nearby unconstrained sites (yellow) breaking the second face condition by entering the vertex spheres (blue) of the constrained sites (grey). The resulting constrained ridges are no longer directly connected.

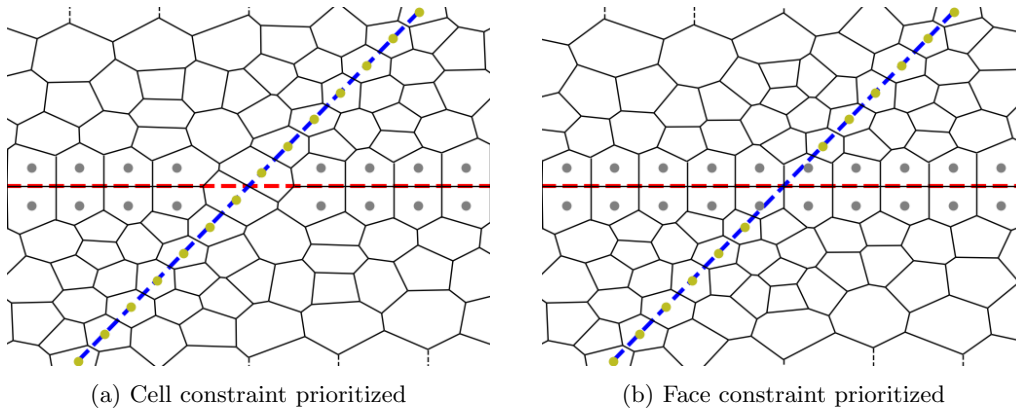


Figure 2.8: * An intersection between a face constraint and a cell constraint, handled by prioritizing one of the two constraints. The sites constrained by the face constraint (red line) are shown as grey points. The sites constrained by the cell constraint (blue line) are shown as yellow points.

2.3 Cell constraints

The UPR module [4, 18, 5] also introduces cell constraints. Cell constraints place direct restrictions on the site locations themselves, forcing them to follow a given line. Its main purpose in UPR is to represent wells in reservoir simulations. While important, cell constraints will not be considered in this thesis. Both the implementation and use cases are comparatively simpler than the face constraints. Additionally, cell- and face constraints are often not possible to fulfill simultaneously, since face constraints require pairwise placements of sites on opposite side of the given face, while cell constraints require them to follow a given line. In the case of an intersection between the two, the sites cannot be mirrored across the face constraint and follow the cell constraint line simultaneously unless the cell constraint line is orthogonal to the face constraint, see Figure 2.8. This means at one of the constraints must yield to the other.

Chapter 3

Method

The following chapter will present the development of constrained Voronoi meshing in \mathbb{R}^3 . The purpose of this thesis is to develop an open-source software package capable of generating constrained Voronoi meshes in \mathbb{R}^3 . The implementation is designed to utilize Gmsh as a mesh generator for the distribution of sites and vertices. The implementation will avoid making modifications to Gmsh itself and will instead focus on utilizing its existing features for Voronoi mesh generation. This both simplifies the implementation and makes the methods applicable to other mesh generators.

3.1 Subdividing face constraint

A face constraint in \mathbb{R}^3 is defined by a corresponding surface. In this thesis these consist of planar faces. As explained in Section 2.1.3, the method to embed the final mesh with these surfaces is divided into two steps: Subdividing the face into target ridges and assigning each ridge vertex a radius. If the vertices and radii follow the face conditions, see Definition 5 and 6, the resulting intersection of the vertex spheres will give us our site locations; see Figure 2.6.

Each ridge correspond to a single pair of sites. As such, all the vertex spheres of a target ridge must intersect at two locations. We know from triangulation that a pair of points in \mathbb{R}^3 are uniquely defined by the distance to 3 points. As such, if the target ridges have more than 3 vertices, at least one of the vertex spheres will have a restricted radius.

In order to resolve stray sites we wish to be able to move the sites by changing the vertex radii. As such it is crucial that the vertex radii are unrestricted. We therefore cannot divide the surfaces into ridges with more than 3 vertices. This restricts us to using triangles for the surface subdivision. As a result, while the interior of the Voronoi meshes will contain polygonal cells, the faces bordering the constraint surfaces will be triangular.

3.1.1 Polygonal constraint borders

While it is not possible to use the vertex sphere intersection method for polygonal ridges, it is possible to achieve similar results through a different approach that allows for polygonal edges. The benefit of the vertex sphere approach is that it works for curved surfaces. However, if the constraint surfaces are planes, a simplified method can be utilized.

Placing pairs of sites equidistantly from some planar surface results in the surface being embedded in the final mesh; See Figure 2.3. While there is still a requirement that the closest sites to the constraint face are the constraint sites, the restriction of shared vertices is no longer present, allowing for polygonal ridges along the constraint surface in \mathbb{R}^3 .

A naive implementation using this method was attempted by Amenta and Bern [2]. This method

struggled with sporadic small ridge segments where the normals of the ridge would differ wildly from the normal of the constraint. However, this was primarily due to the curvature of the constraint surface. In theory, an algorithm could be implemented that creates Voronoi meshes with polygonal constraint borders, given the constraints are planar.

In this thesis however we will focus on the vertex sphere method utilized by UPR[4] and `Vorocrust` [1]. While we will mainly focus on the implementation of planar constraint surfaces, the vertex sphere method is more generally applicable and open for further development.

3.2 Designating vertex sphere radius

Once constraints have been subdivided into ridges and vertices, the radius of each vertex sphere needs to be decided. A valid set of radii follows the two face conditions; see Definition 5 and 6. First, the spheres centered on the vertices of any given target ridge must intersect. The reason for this is self-explanatory, as we intend to place sites at the intersections. This essentially places a lower and upper bound on the size of a given vertex sphere based on the radii of its immediate neighbours; see Figure 4.3 and 4.4. Secondly for any given vertex, its sphere must not contain the sites of any non-neighbouring ridge. We refer to any site breaking this condition as a stray site.

If any stray site lies within the interior of the vertex sphere, it ends up being closer to the target vertex than the intended neighbouring sites. This effectively moves the resulting vertex away from its target position and can also drastically change the ridges bordering the vertex. Thus for the face constraints to be upheld, we need to prevent any stray site from being introduced.

While the position of the sites are restricted by the vertex spheres, the vertex sphere radii are not themselves restricted aside from their minimum and maximum value. As such, we may move stray sites out of conflicting vertex spheres by shrinking or increasing the size of the spheres.

3.3 Making centroidal cells

In addition to determining a valid configuration of vertices and radii that satisfy the face constraints, it is also desirable for the resulting cells to exhibit good geometric properties. As mentioned previously, a crucial metric for assessing the quality of the mesh is the degree to which the site locations align with the centers of their respective cells. Although we will not directly optimize this criterion, it provides valuable insights into the desired cell characteristics. Specifically, we aim to avoid elongated cell shapes. This consideration holds significance for both the subdivision process and the assignment of radii. The assignment of radii is primarily responsible for the occurrence of elongated cells, as the cells tend to be well-formed when the vertex radii are proportional to the ridge sizes. However, during the subdivision step, there is a possibility of encountering vertex positions where all permissible sets of radii result in distorted cell shapes

To mitigate this issue, it is crucial to ensure that the discretization step generates a triangulation that facilitates the identification of an easily attainable radius solution. Discretizations characterized by sporadically varying triangle shapes not only complicate the process of finding a suitable radius solution but also lead to more peculiar outcomes. As the determination of the radius solution involves iterative adjustments, such irregular triangulations tend to deviate further from the initially set radii, which are intended to approximate an ideal state. Thus, it is desirable to avoid such deviations by promoting a more uniform triangulation structure.

To achieve the generation of centroidal cells, it is essential to establish the relative ratios between certain scalar quantities associated with a given ridge. These quantities include the mesh size denoted by d , representing the length of the ridge edges, the vertex radii denoted by r , and the distance denoted by h between the ridge and its corresponding sites. While these quantities may vary depending on the size of the ridge, it is desirable to maintain consistent relative sizes among them. These ratios are based on an ideal scenario where the target ridge is perfectly equilateral. Although the tessellated triangles in practical applications tend to exhibit variations in shape,

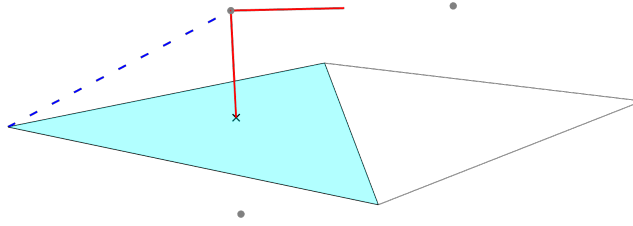


Figure 3.1: The site height shown as a red line is determined in relation to the ridge edge to ensure that it maintains an equal distance from the mirrored site and neighboring sites on the same side of the constraint. The resulting vertex radius is depicted in blue.

high-quality meshes tend to approximate equilateral triangles. The elected ratios are illustrated in Figure 3.1

We would like the constrained sites to be equidistant from one another. This means that the site height from the ridge surface h needs to be equal to the distance from the center of the equilateral triangle to its edges. This gives us a relation between the height and the mesh size. The relation between the radii of the vertex spheres is simply given by the distance from the corner to the site.

This results in a set of ideal relations between the cell sizes. Currently these are set in relation to the mesh size, as the mesh size is a general standard often utilized by meshing algorithms to describe the size of cells. The resulting ratios are shown in Table 3.1.

Variable	Mesh size(d) ratio	Description
Mesh size(d)	—	Size of shortest edge in cell
Site height(h)	$d \frac{1}{2\sqrt{3}}$	Distance from constrained site pair to their dividing ridge
Vertex radius(r)	$d \frac{\sqrt{5}}{2\sqrt{3}}$	Radius of the vertex spheres

Table 3.1: Ratios between cell sizes

Chapter 4

Development

The goal of this project is to create an algorithm capable of subdividing a set of face constraints into triangular surfaces and assigning each vertex a valid radius, defined by the face conditions outlined in Section 2.1.3. There are many different approaches one can take to achieve this. `Vorocrust` [1] uses an iterative method in which the radii of the spheres are shrunk to fit a set of criteria. When this in turn results in missing coverage of the faces, additional vertices are introduced as needed. Eventually this method is guaranteed to converge to a valid set of vertices and radii, given enough iterations.

In this thesis, we will utilize a different approach. We would like to be able to perform one initial subdivision of the constraint faces without the need to introduce additional vertices or triangles down the line. This means that a valid set of vertex radii should exist for the initial subdivision of the constraints. This set of radii should additionally result in somewhat centroidal cells. The main benefit of this approach is its ease of implementation into existing mesh generators. Since our approach will not rely on recursively adding additional vertices if the Voronoi generation fails, we can implement the method without needing to modify the meshing algorithms themselves.

We will utilize Gmsh[8, 9], an open-source mesh generator to distribute points in \mathbb{R}^2 and \mathbb{R}^3 . Gmsh is capable of constructing high-quality triangular/tetrahedral grids, making it ideal for use in Voronoi mesh generation. It will be utilized for two purposes: the subdivision of the face constraints and the distribution of unconstrained sites. Gmsh allows the specification of planes and volumes to be filled in using a selection of different meshing algorithms. It works by first defining a boundary, such as a loop in 2D or the surface boundary of a 3D volume. It then discretizes the space according to a given mesh size field. This field describes the mesh size for any point in \mathbb{R}^3 and can have varying levels of customization, from pre-made density functions to custom mathematical expressions.

4.1 Baseline implementation

Initially, a naive approach was attempted on a test case. The test case consisted of a unit cube with two intersecting planes in its interior. The purpose of this test was to see how well-formed the cells would be when using a uniform mesh density for the initial subdivision of the face constraints.

The method consisted of three distinct steps. The constraint faces were first divided into triangles using Gmsh with a constant mesh density; see Figure 4.1. The vertices in the triangles generated from this would then be utilized as the target vertices for the remaining steps.

Afterwards, an initial radius was designated to each vertex sphere. This initial radius was based on the mean distance to the neighbouring vertices; see Table 3.1. Since the triangulation divides the surfaces into approximately equilateral triangles, these initial radii should result in centroidal cells, so we would like to avoid modifying them whenever possible.

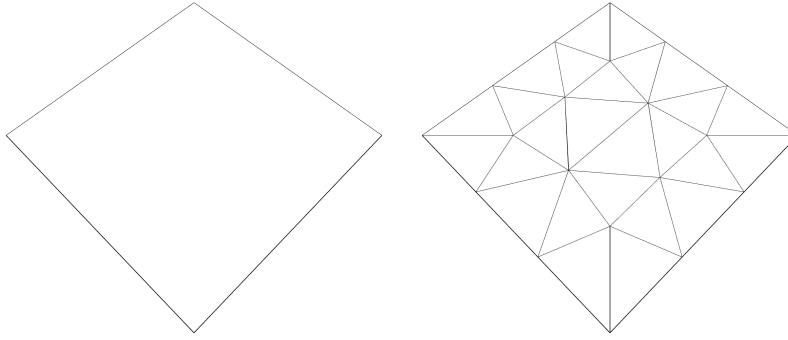


Figure 4.1: Unit plane triangulated using Gmsh.

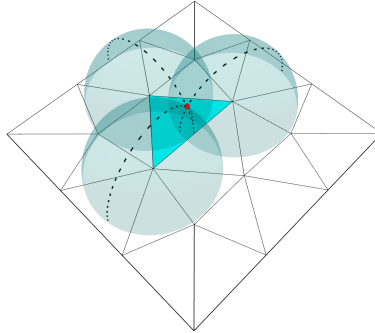


Figure 4.2: The sites corresponding to a given triangle, highlighted in blue, are placed at the intersection of the three spheres centered at the triangle's vertices. Each vertex has only one sphere radius, used in the intersections of all the neighbouring sites.

After designating the initial radii, a minimum radius is assigned to each vertex based on the radii of its neighbours. For a given vertex sphere, this is the smallest radius for which its sphere still makes intersections with all its neighbouring spheres; see Figure 4.3. Since we can only get the position of a triangle's sites if its three vertex spheres intersect, we cannot shrink any vertex sphere beyond its minimum radius.

A preliminary set of sites is then constructed from the initial vertex positions and radii. If there are any stray sites, we attempt to remove them by shrinking the sphere radii of the balls. For every vertex containing a stray site, the sphere of the vertex itself is shrunk. Additionally, to prevent lopsided cell shapes, we also shrink the radii of the three vertices whose intersection form the stray site. This aims to move the stray site and the conflicted sphere away from each other.

Finally, once there are no longer any stray sites, the volume is filled with unconstrained sites. Using Gmsh, the volume specified by the boundary faces is filled with so-called background sites to ensure an even density of cells across the space. These sites are less constricted by the face constraints. The only requirement is that they do not enter the constraint vertex spheres. However, since these sites are not necessary to enforce any of the constraints, we can simply remove any stray background site.

While the initial plan was to evaluate the mesh quality of such a naive approach, the test was entirely unsuccessful, failing to even produce a valid mesh. The algorithm would often end up in a situation where in order to prevent a stray site, the radii of some vertex sphere had to be reduced beyond their minimum value. Different approaches to sphere shrinking were attempted to fix this issue. The stray sites can be moved in any direction by increasing or decreasing the radii of their three spheres. As such, different directions of movement were attempted to remove the stray from the conflicted sphere.

The first attempt was to choose the direction in which the three spheres shrink proportionally to

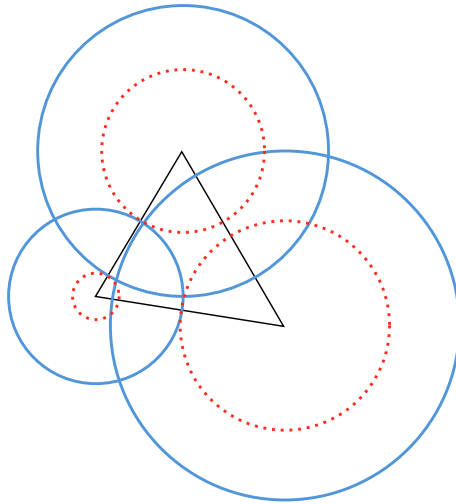


Figure 4.3: The minimum radii of the vertices of a triangle seen from above. The blue circles represent the intersection between the vertex spheres and the triangle plane. The red circles show the minimum radius of each vertex sphere. If any of the sphere radii are shrunk beyond these minimum values, the spheres no longer intersect at one point.

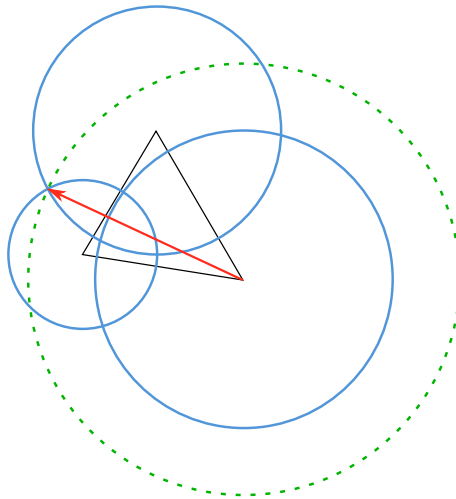


Figure 4.4: The maximum radii of a vertex seen from above. The blue circles represent the intersection between the vertex spheres and the triangle plane. The red arrow illustrates the maximum radius of one of the vertices. If this radius is exceeded, the sphere will encompass the entire intersection circle formed by the other spheres, instead of intersecting it.

the difference between their current and minimum radii. This was done to allow the stray to move further without being limited as much by a single minimum vertex radius. The second attempt was to move the stray site radially away from the conflicted vertex, as this could resolve the strays with less total radius shrinkage. The last attempt was to move the stray site towards the circumcenter of its corresponding triangle. For any acute triangle, this is the point in the interior of the triangle where the minimum distance to any of the vertices is largest.

It is in general difficult to tell what the ideal direction to move a stray is; the choice of direction is a greedy approach to a nonlinear optimization problem. None of these approaches seemed to make a significant difference, with the same number of unsolvable stray sites occurring regardless of the direction chosen.

The problem was partially solved by reducing the proportionality constant of the initial radii. This in turn resulted in narrow, misshapen cells that barely managed to form a valid Voronoi mesh. The selection of proportionality constant also required several attempts before resulting in a valid mesh. If the constant was set too low, some vertices would end up with initial sphere radii that could not cover all its neighbours. Setting the initial radii using the maximum edge distance instead prevented this, but did not result in a radius set without strays. The algorithm only produced a valid set when the proportionality constant was within a small range, only when using the median edge distances, and produced sub-optimal cells.

While it is clear from experimentation that a valid set of radii could be found for the initial triangulation, the naive approach was not sufficient to find such a valid set consistently. Utilizing the mesh generator required manual adjustment of parameters that should ideally be constant.

There are essentially two areas where we can improve our approach: the subdivision step and the radii determination. It is apparent that improving the radius determination can resolve the issue, as we have shown a valid set of radii exists. There are several improvements that can be done in this regard. The minimum radius of certain vertices can be reduced by increasing the radii of the immediate neighbours. This would allow more contested spaces to have smaller vertex radii.

Alternatively, the nudging direction of the strays could be managed more dynamically. In many cases a stray site can be prevented without significantly reducing the sphere radii of its associated spheres. This is especially noticeable when the radii of the vertices are large compared to the distance between them. For instance, moving the stray tangentially to the center of its associated triangle will typically affect the radii less than nudging it radially.

Trying to solve the issue with better radius designations has its drawbacks. First of all, it is massively more computationally difficult. The problem of finding a valid set of vertex radii is a constrained optimization problem with nonlinear constraints. When constructing meshes, the number of vertices are often in the thousands. The number of constraints scales by the square of the number of vertices, since every site must avoid the sphere of every non-neighbouring vertex. Thus the non-linear constrained problem ends up having thousands of variables and millions of constraints, which is not computationally feasible. To keep the algorithm performant, an iterative solution seems the best option. There are of course many modifications that can be made to how the iterative function works, but most of the design choices serve a purpose.

Most of the the decision to only shrink vertex spheres rather than allow them to be increased is to prevent the introduction of additional strays. Shrinking a vertex radius typically will not introduce new stray sites. In addition to making the sphere smaller, it forces the sites of connected triangles to be closer to the triangle plane, which in itself helps avoid strays. While there are cases where increasing vertex radii prevents strays, it is likely to introduce new strays. A method that allows for radius increment then has to recursively prevent new strays introduced. It also runs the risk of getting stuck in a loop where it introduces more strays than it solves at each iteration. When the algorithm only shrinks spheres, we can deal with strays one vertex at a time, with the assumption that the algorithm will not introduce strays to previous vertices.

Furthermore, restricting the way valid radii are found also serves the purpose of maintaining proper cell shapes. The initial radii may be naive, but they are also ideal when it comes to making the constrained cells centroidal. As a result, keeping the radius selection simple is largely beneficial in

terms of performance, ease of implementation and final mesh quality.

Thus, we turn our focus to improving the initial triangulation. The aim is to generate a set of target ridges for which our iterative radius approach will be sufficient. Ideally, we would wish for the initial designated radii to require no further modifications. If the radii of the vertex sphere remain close to their initial values, the resulting cells will be close to centroidal and the mesh quality will be higher.

4.2 Causes of stray sites

To improve the initial subdivision of the face constraint, we need to establish what conditions cause stray sites to occur. From the test case, we noticed that the strays typically occurred in areas where the spacing between constraints were narrow, such as corners. What constitutes two face constraints being close depends on the size of the cells; smaller cell sizes typically resolve the issue of close face constraints. It should be possible to locally scale the cell sizes in areas where the constraints are close. However, the concept of close in this context is not simply the distance between the constraints.

In an ideal setting, a pair of sites is placed along the normal of the face constraint, with a distance that is proportional to the cell size. We see that for a given cell density, this essentially results in a volume on either side of the face constraint wherein all sites are located. Stray sites are generated in the overlap of at least two such volumes. Thus, we may shrink the cell size to prevent these overlaps and stray sites from occurring.

The volumes extend normally from both sides of any given face constraint. As such, the maximum distance before intersection can be determined for a point on the given face as the maximum distance to another face site boundary. We assume that the width of the other boundaries are decided similarly, meaning the intersecting boundary will at most have a width equal to the width currently being calculated. As such, we propose a new distance function, hereby named the **inscribed sphere distance**.

4.3 The inscribed sphere distance

In this section we wish to create a function that can evaluate the mesh size at every point on the constraint we wish to triangulate. We aim to avoid strays by reducing the mesh size of the constraint triangulation in more contested areas. We refer to the given point as the origin point \vec{p}_o and the surface as the origin surface or plane. This density function is defined by a set of constraint surfaces just like the one we were trying to discretize, and we assume that the same density function is utilized to determine their mesh density.

An obvious candidate for the mesh density function would be the Euclidean distance. If the radius of a vertex at any point on a constraint is limited to be less than half the smallest distance to any another constraint, the vertex spheres of opposing constraints would never overlap, meaning no stray sites. However, this function fails to evaluate the density necessary to prevent stray sites coming from the same constraint. When using Euclidean distance we obviously cannot include a constraint surface in its own distance evaluation, the distance is always zero. It is, however, possible for the curvature of the constraint itself to introduce stray sites. The troublesome issue with the vertex spheres is how we only want some spheres to intersect.

Additionally, two constraint being close in Euclidean distance does not necessarily mean they are at risk of forming stray sites. The sites generally propagate normally from the constraint surfaces. As such, two constraints with opposing normals will generally require smaller mesh sizes to avoid the introduction of strays. The distance function should not only consider the distance between the constraint, but also the alignment of their normal.

One such function is the proposed **inscribed sphere distance**. For a given origin point \vec{p}_o ,

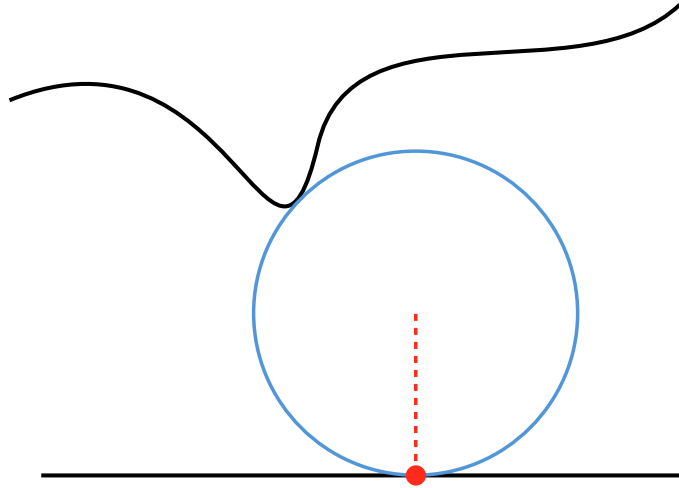


Figure 4.5: Inscribed sphere distance for a given point, shown in red. The direction from the point to the sphere center is the normal of the plane at the position of the point.

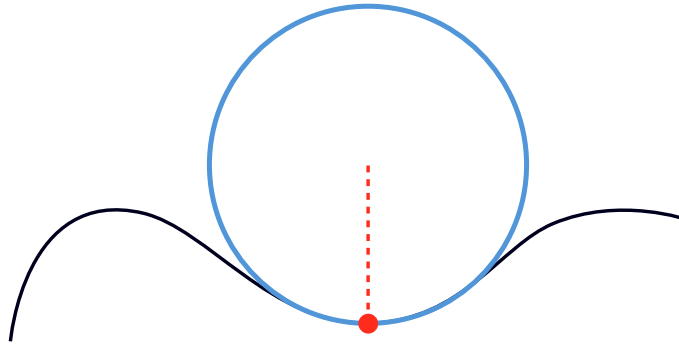


Figure 4.6: Inscribed distance being limited by the curvature of the surface

direction \vec{n}_o and a set of constraint surfaces, the inscribed sphere distance is defined as the radius of the largest sphere that tangents the given point and whose center lies in the given direction from the point without intersecting or containing any of the constraint surfaces; see Figure 4.5. For a set of possible directions $\vec{n}_o \in N$, the ISD is defined as the minimum ISD of all the possible directions.

We define the set N as the directions orthogonal to the surface or line \vec{p}_o lies on. This surface or line is referred to as the origin surface/line. For a plane this means N consists of the positive and negative of the plane normal $N = \{\vec{n}_s, -\vec{n}_s\}$. This is because we expect the sites belonging to any constraint to be placed orthogonal from the constraint surface.

The ISD takes the alignment of the constraints into consideration. If the surface normals of two constraints oppose one another, the resulting inscribed sphere. Likewise, two constraints on the same plane will have an infinite ISD, since regardless of the mesh size, the sites of one constraint will not interfere with the vertex spheres of the other.

Additionally, the ISD takes into consideration the curvature of the constraints. The inscribed sphere needs to avoid intersecting the origin plane, limiting the ISD for curved surfaces. This in turn prevents self introduced stray sites from curved surfaces.

We note that the ISD has to take into consideration both sides of the origin plane \vec{p}_o , since the sites will be placed equidistantly on either side of it. Since the constraint surfaces may intersect the origin plane, the ISD must be considered for both sides of the origin plane.

The ISD for any set of face constraint is the minimum of the inscribed distance to any individual

face constraint. Since we will be dealing with a constraint set consisting of triangular faces, we may calculate the ISD to each triangle individually and return the smallest value. We therefore need to define the inscribed distance between a point on a plane and a triangle.

If the given point is in the interior of a plane, we see that the only way for the sphere not to contain points on the plane is for its center to be placed in the direction of the face normal. Similarly, if the point is on the edge of a face constraint, the center needs to be placed orthogonal to the edge direction. If the point is on a corner of a face, the sphere center can be in any direction from the given. This allows us to generate mathematical expressions of the ISD between two triangles.

4.3.1 Surface to triangle ISD

The ISD for a point on a surface to a given triangle is a combination of three different functions. We notice that the distance to a given unbounded plane is always less than or equal to the distance to a line on the plane. This in turn is less than or equal to the distance to any point on the plane. Since we want the minimum distance, this means the distance to the interior of the triangle takes priority over the distance to the edges, which again takes priority over the distance to the triangle vertices. However the sphere will only be tangential to the triangle interior or edge interiors within given areas. As such we need to define both the three distance functions as well as the criteria for when the functions are applicable.

First, we define the plane to plane ISD; see Figure 4.7. The distance from any given point to a plane is a linear expression $d = \vec{n} \cdot \vec{p} + D$, assuming $\|\vec{n}\|_2 = 1$. Since the distance from the origin point \vec{p}_o to the sphere center is equal to the distance between the plane and the sphere center, we can redefine the sphere center p as a function of the origin point \vec{p}_o , the origin normal \vec{n}_o and the inscribed circle distance d , $\vec{p} = \vec{p}_o + d \cdot \vec{n}_o$. The expression then becomes

$$\begin{aligned} d &= \vec{n} \cdot (\vec{p}_o + d \cdot \vec{n}_o) + D \\ d(1 - \vec{n} \cdot \vec{n}_o) &= \vec{n} \cdot \vec{p}_o + D \\ d &= c \cdot (\vec{n} \cdot \vec{p}_o + D) := \vec{n}_c \cdot \vec{p}_o + D_c \end{aligned} \tag{4.1}$$

This results in another linear expression proportional to the original planar distance expression, with proportionality constant $c = (1 - \vec{n} \cdot \vec{n}_o)^{-1}$. This proportionality constant goes toward $1/2$ as \vec{n} approaches $-\vec{n}_o$ and towards infinity when \vec{n} approaches \vec{n}_o , becoming undefined when they completely align. The reason for this is that the expression assumes the inscribed sphere is on the positive sides of both the triangle plane and the origin plane. Since the inscribed sphere can be on either sides of both planes, we must introduce some additional condition.

First, we notice that the inscribed sphere by definition must have its center on the same side of the plane triangle as the origin point, otherwise the sphere would contain parts of the triangle. As such, we invert the triangle normal in Equation (4.1) if the origin point is on the negative side of the triangle.

For a given triangle plane, the ISD will always be smaller on the side where $\vec{n} \cdot \vec{n}_o$ is smallest. However, as we will explain shortly, the inscribed sphere does not necessarily contact the interior of the triangle on both sides of the origin plane. For this reason, the ISD must be calculated twice, for \vec{n}_o and $-\vec{n}_o$.

The plane ISD defined only applies if the point of contact with the triangle plane is inside the triangle. We can check whether this is the case by checking if the contact point is on the positive side of three border planes spanned by the triangle normal and edge directions. Since the border planes are orthogonal to the triangle normal, the contact point is in the triangle interior as long as the sphere center is inside the three planes. The position of the sphere center is already defined by an affine transformation of the origin point. Thus we transform the border expressions into three linear expressions on the origin plane; see Figure 4.8. If all three expressions evaluate to positive values for a given point on the origin plane, the sphere will border on the interior of the triangle and the ISD will be given by equation (4.1).

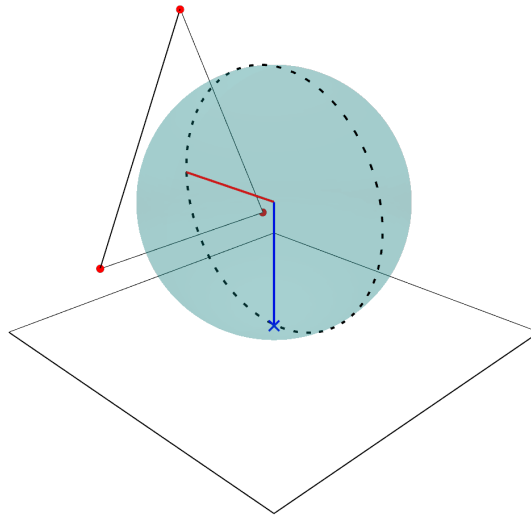


Figure 4.7: Plane-to-plane ISD: Inscribed distance between a point on a plane and a triangle when the sphere tangents the interior of the triangle. The blue and red lines are the normals of the plane and triangle, respectively

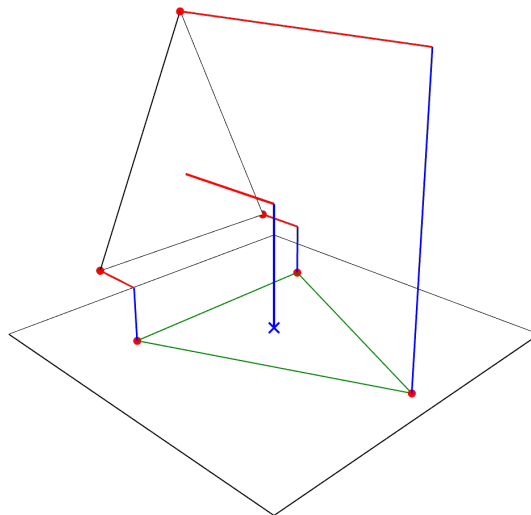


Figure 4.8: In the interior of the boundary, shown in green, the points on the plane will have inscribed spheres that tangents the interior of the constraint triangle.

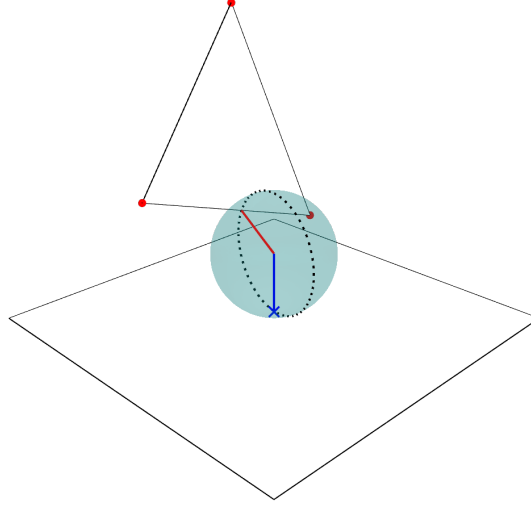


Figure 4.9: Plane-to-line ISD: between a point on a plane and a line segment.

In the cases where the point of contact is not in the interior of the triangle, it is either on one of the triangle edges or corners. Each of the triangle edges has a corresponding face border on the origin plane, see Figure 4.10. The point of contact may only lie on an edge if the origin point is outside its corresponding face border. To further decide if the inscribed sphere borders the edge or a point, we define the ISD for the edges.

Using the vector formulation for a line,

$$l := \{x, x = \vec{p}_l + t\vec{n}_l, \|\vec{n}_l\|_2 = 1, t \in \mathbb{R}\},$$

The distance from a given point \vec{p} to the line is given by

$$d(\vec{p}, l) := \|(\vec{p} - \vec{p}_l) - \vec{n}_l((\vec{p} - \vec{p}_l) \cdot \vec{n}_l)\|_2.$$

Once again, using the fact that the origin point and contact point are equidistant from the center of the sphere, we get $\vec{p} = \vec{p}_o + \vec{n}_o \cdot d(\vec{p}, l)$, resulting in the equation

$$\begin{aligned} d &= \|(\vec{p}_o + d\vec{n}_o - \vec{p}_l) - ((\vec{p}_o + d\vec{n}_o - \vec{p}_l) \cdot \vec{n}_l)\vec{n}_l\|_2 \\ &= \|d(\vec{n}_o - (\vec{n}_o \cdot \vec{n}_l)\vec{n}_l) + (\vec{p}_o - \vec{p}_l) - \vec{n}_l((\vec{p}_o - \vec{p}_l) \cdot \vec{n}_l)\|_2 \\ &= \|d\vec{a} + \vec{b}\|_2, \\ \vec{a} &= \vec{n}_o - (\vec{n}_o \cdot \vec{n}_l)\vec{n}_l, \vec{b} = (\vec{p}_o - \vec{p}_l) - \vec{n}_l((\vec{p}_o - \vec{p}_l) \cdot \vec{n}_l). \end{aligned} \tag{4.2}$$

We square both sides to get the quadratic equation

$$\begin{aligned} d^2 &= d^2\|\vec{a}\|_2^2 + 2d(\vec{a} \cdot \vec{b}) + \|\vec{b}\|_2^2, \\ d &= \frac{-(\vec{a} \cdot \vec{b}) \pm \sqrt{(\vec{a} \cdot \vec{b})^2 - \|\vec{b}\|_2^2(\|\vec{a}\|_2^2 - 1)}}{\|\vec{a}\|_2^2 - 1}. \end{aligned} \tag{4.3}$$

This results in two possible ISD values, one positive and one negative. This follows from the fact that any line not orthogonal to the origin normal will eventually pass through the origin plane. As such, there are two possible inscribed spheres, one on either side of the origin plane. Since we are interested in the smallest distance, we take use the smallest absolute value of the two, as long as the contact point is on the edge; see Figure 4.9.

The edges are not unbounded lines, but line segments forming the triangle. Because of this we need to see if the two points of contact are inside the edge interior; see Figure 4.10. This is accomplished by comparing how far along the given edge direction \vec{n}_l the point of contact is. We then compare

this value to the same value calculated for the start and end positions, \vec{x}_0 and \vec{x}_1 , of the edge. The point of contact is inside the edge if the following equations hold

$$\vec{x}_0 \cdot \vec{n}_l \leq \vec{p}_c \cdot \vec{n}_l \leq \vec{x}_1 \cdot \vec{n}_l$$

While we do not yet know the point of contact, we know that the vector between the contact point and the center of the inscribed circle is orthogonal to the line direction, we can substitute the point of contact \vec{p}_c for the inscribed circle center \vec{p} in the condition, resulting in the new equation,

$$\vec{x}_0 \cdot \vec{n}_l \leq \vec{p} \cdot \vec{n}_l \leq \vec{x}_1 \cdot \vec{n}_l.$$

In the case where the point of contact is neither in the interior of the triangle or on one of its edges it must be on one of the three triangle vertices. The distance to a given vertex is simply the Euclidean distance between the point and the vertex. By defining the point p as in the previous equations, we get.

$$\begin{aligned} d(\vec{p}, \vec{p}_v) &= \|\vec{p} - \vec{p}_v\|_2 = \|\vec{p}_o + d\vec{n}_o - \vec{p}_v\|_2, \\ d^2 &= \|\vec{p}_o - \vec{p}_v\|_2^2 + d^2\|\vec{n}_o\|_2^2 + 2d((\vec{p}_o - \vec{p}_v) \cdot \vec{n}_o). \end{aligned} \quad (4.4)$$

Since $\|\vec{n}_o\|_2 = 1$, the d^2 terms cancel and the equation becomes:

$$2d((\vec{p}_o - \vec{p}_v) \cdot \vec{n}_o) = -\|\vec{p}_o - \vec{p}_v\|_2^2, \quad d = \frac{-\|\vec{p}_o - \vec{p}_v\|_2^2}{2(\vec{p}_o - \vec{p}_v) \cdot \vec{n}_o} \quad (4.5)$$

In the case where d is negative, the point is on the opposite side of the origin plane. Since the inscribed sphere has to contact one of the triangle vertices without containing any of the others, the inscribed circle distance is taken as the minimum of absolute distances to the vertices.

The resulting ISD from a plane to a given triangle is a continuously differentiable function; see Figure 4.11. However, when the minimum of the triangle ISD is taken, the resulting expression is no longer differentiable at the junction between two triangles.

One noteworthy property of this implementation of the ISD is that it is unique to each origin plane. As such, the function is not meant to be a three dimensional volumetric field. The field is defined by a given plane used as an origin plane and is only correct for points on the corresponding plane, despite being defined for all points in \mathbb{R}^3 . One downside to this is how typically FEM mesh generators, including Gmsh, generate the entire mesh at once. This requires the density function to be defined for all faces simultaneously. In our case, this is solved by meshing in the faces individually to then combine the generated faces in a separate step before the volume is filled in. This does, however, slightly affect performance, as Gmsh has to initialize several times, see Chapter 6.

4.3.2 ISD cell proportion

Now that we have a definition for the ISD, we need to relate it to the remaining sizes outlined in Section 3.2. We therefore need to relate one of the values in Table 3.1 to the ISD. As previously mentioned, to make the cells centroidal, it is preferable that the sites are equidistant to one another. This means the distance between the site pairs mirroring the plane should be equal to the distance between neighbouring sites on the same side of the plane.

This principle also applies to sites associated with different constraints. It is desirable for these sites to maintain a somewhat equal spacing with neighboring sites. To achieve this, we focus on the site height h . Initially, if we set the site height to half of the ISD, the distance between two sites belonging to opposing constraints would be equivalent to the distance between neighboring sites. However, this configuration is not ideal as we aim to allocate sufficient space between constrained sites to accommodate unconstrained sites.

Unlike the cells corresponding to constrained sites, which by construction need to contain a triangular ridge, the unconstrained cells are not restricted and are better suited for junctions between

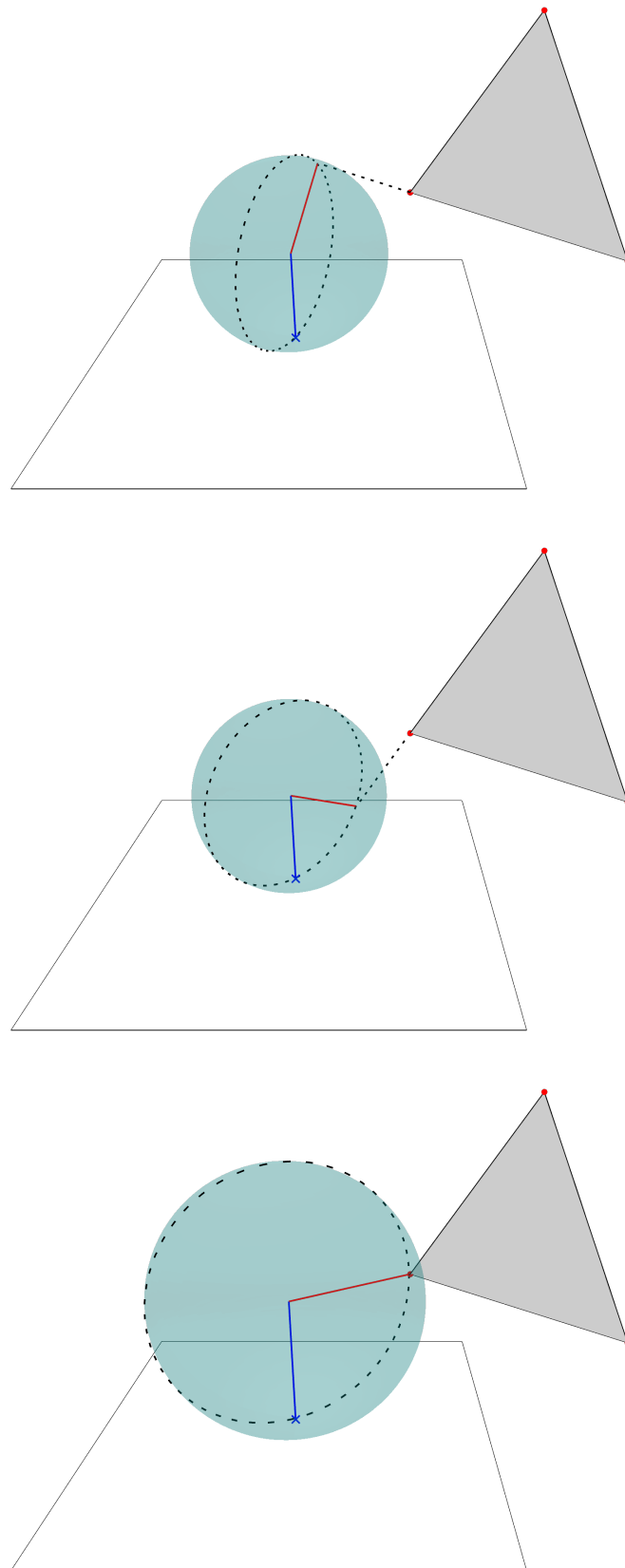
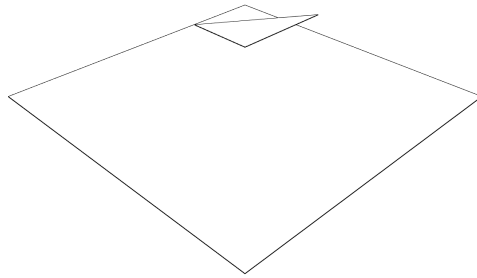
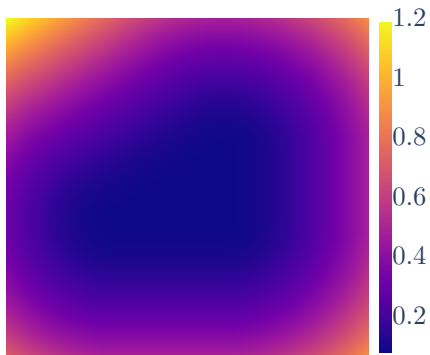


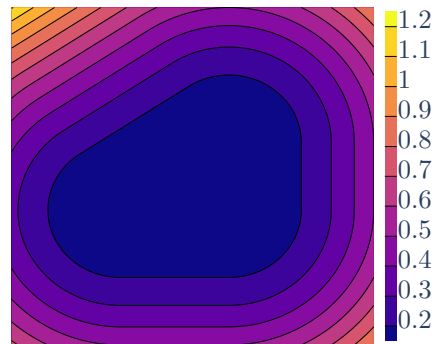
Figure 4.10: The point of contact between the inscribed circle and the two potential triangle lines is outside the ends of the line segment. Consequently the point of contact is on one of the triangle corners.



(a) Test case



(b) Heatmap of the ISD



(c) Contours of the ISD

Figure 4.11: The ISD of a plane is evaluated by considering a single constraint triangle positioned above it. The heatmap visualization illustrates the continuity of the function, while the contour plot showcases the continuous gradient.

two constraint surfaces. Therefore, we assign the site height h as one quarter of the ISD, allowing for additional space within the interior of unconstrained sites. The remaining ratios are determined by their correlation to h as illustrated in Table 4.1.

Variable	Mesh size(d) ratio	ISD(R) ratio	Description
Mesh size(d)	-	$R\frac{\sqrt{3}}{2}$	Size of shortest edge in cell
Site height(h)	$d\frac{1}{2\sqrt{3}}$	$R\frac{1}{4}$	Distance from constrained site pair to their dividing ridge
Vertex radius(r)	$d\frac{\sqrt{5}}{2\sqrt{3}}$	$R\frac{\sqrt{5}}{4}$	Radius of the vertex spheres

Table 4.1: Ratios between cell sizes and the ISD

4.3.3 Preliminary ISD test

To demonstrate and test the ISD distance, we set up a test case consisting of two planes narrowing at one side; see Figure 4.12. The two planes are triangulated using a ISD based mesh element size. The resulting triangulation can be seen in Figure 4.13. We see that the density of triangles

increases towards the narrowing of the planes.

The vertex radii are still decided based upon the mean distance from each vertex to its neighbours. While we could decide the radii directly based on the ISD at the position of the radii, this can more easily cause large variation in site positions. The triangulation approximates the density specified by the ISD, but there is still some intermittent variation in triangle sizes. If we assign the same vertex radii to triangles of different sizes, the resulting sites will be at varying heights from the constraint. We want the vertex radii to be based on the actual size of the triangular elements, rather than their theoretical size. After setting the vertex radii, we get the sites from the intersection; see Figure 4.14.

Using these sites, we generate the Voronoi mesh. To see whether the mesh matches the target, we only plot the ridges bordering the constraints; see Figure 4.15. We also truncate the side of the mesh to see the interior cell shapes.

We see that the ISD-based mesh density has accomplished what we are looking for. The dynamically increased triangle density prevents stray sites from occurring without the need to drastically modify any radii. We see that the resulting cells are well-shaped, with their width matching their height.

One issue that becomes apparent from the test case is that of intersecting face constraints. As shown in Figure 4.13, the size of the triangles is proportional with the ISD. At an intersection between two face constraints, the ISD is zero, meaning the mesh density becomes infinitely high as we approach the intersection. This is an intended feature, as the density is constantly increasing to prevent overlap between the sites of an infinitesimal narrowing. However, since we cannot have infinite mesh density, we need a way to handle the intersections.

4.4 Sharp edges

Since triangulation using ISD as element size breaks at the intersection of constraints, we need to make a padding around these sharp edges. By this, we mean separating the plane into a padding and an interior space. The interior space is triangulated using ISD as previously demonstrated, while the padding area is responsible for the transition from the ISD density to some non-infinite density at the intersection.

We will refer to the vertices shared by the interior and the padding as the border vertices. Since we want these padding vertices to seamlessly blend in with the interior vertices, we require them to have sphere radii decided by the ISD. We also expect their distance to neighbouring border vertices and interior vertices to be decided by the ISD to match the interior triangle density. This restriction does not however apply to the distance between bordering border vertices and the remaining padding vertices.

We are free to choose the width of the padding, though in practice the choice of width does not make much of a difference in how we transition to the mesh size at the intersection. The reason for this is that the density and radii of the border vertices scale linearly with the padding width. This makes the ratio between the padding width and the element density constant. As we will see later, this essentially allows us to freely decide the width of the padding based on how small we want the border elements to be.

4.5 Minimum size padding

One potential solution would be to assign a minimum mesh size to the padding triangles. This would produce a padding of constant mesh size around each intersection. To compensate for the narrowing of the space between the face constraints, the radii of the vertex spheres would be shrunk to bring the sites closer to the triangles. Here we encounter a problem.

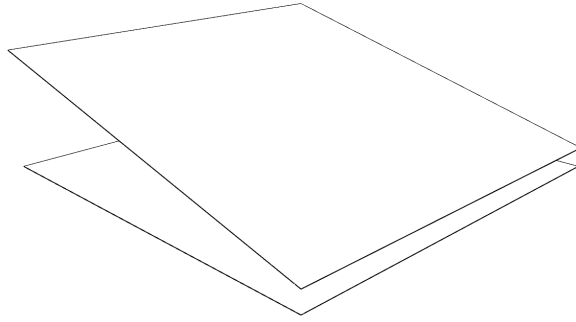


Figure 4.12: Test case for ISD-based density, consisting of two planes that gradually converge.

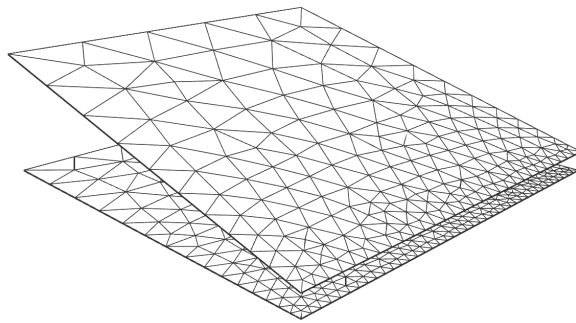


Figure 4.13: Triangulation of the test case using ISD-based mesh density.

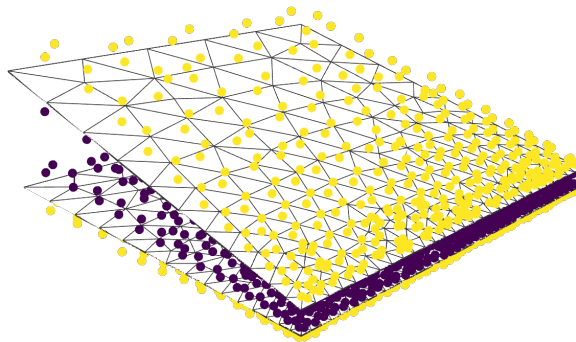


Figure 4.14: Triangulation of the test case with site locations inside(purple) and outside (yellow) the boundary

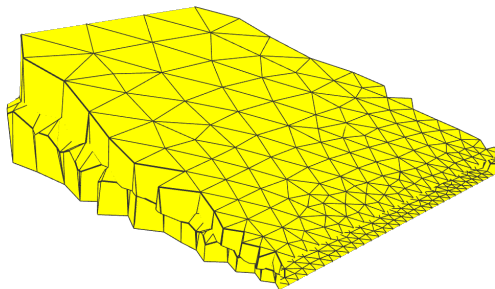


Figure 4.15: Voronoi mesh generated from the test case site locations.

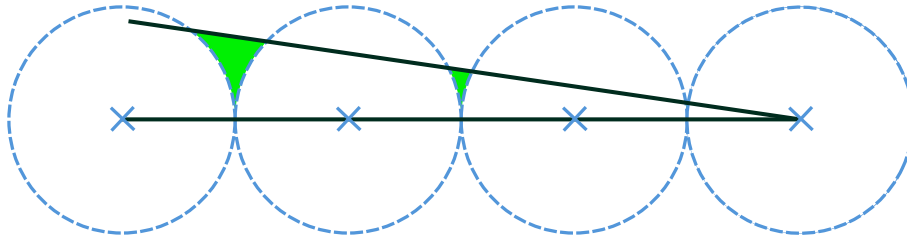


Figure 4.16: When using a constant mesh size, the gradual narrowing between two intersecting planes result in highly contested space. Here the vertex spheres of one constraint, shown in blue, have the smallest radius possible for the constant spacing. The possible spaces the remaining sites can be placed is highlighted in green. Despite the small radii, the available space is very restricted.

As mentioned previously, each vertex essentially has a minimum radius required to intersect with neighbouring vertices; see Figure 4.3. When we shrink the radii of a given vertex, we simultaneously increase this minimum radius of its neighbours. Doing this for all vertices simultaneously results in a gradually more constricted problem, where the current radii of the vertices are increasingly close to their minimum radii. When trying to remove stray sites, this gives less leeway when modifying the vertex radii, and more easily results in situations with no viable solution.

This is a significant problem, as the minimum mesh size approach will also result in more stray sites to begin with. We can only reduce the sphere radii to approximately the distance between the vertices and the triangle center. When two intersecting face constraints are close, this makes it so that each vertex sphere contests a large part of the narrow space between the constraints. A 2D visualization of this is shown in Figure 4.16. It is significantly more likely that a given site will intersect with a vertex from the opposing constraint under these conditions. To summarize, if we triangulate the padding using a minimum mesh size, the resulting border will have more stray sites to remove that must be solved under stricter radius constrictions.

It is worth noting that when the intersecting constraints are closer to orthogonal, the issue becomes less severe. The ISD increases more rapidly, resulting in a padding essentially only consisting of a single row of triangles. The ISD increases so rapidly in fact that the distance from the intersection to the center of these triangles is sufficient to allow for more leeway in the radius selection. As a result, the approach would work if the intersection angles are all obtuse enough.

We see that ignoring the ISD mesh size is not a viable solution to the intersection padding. As mentioned previously, this problem persists even if we reduce this minimum mesh size. Reducing the mesh size while still matching the ISD distance at the padding border will result in the padding being less wide. The relation between the padding mesh size and the distance between the constraints will be the same.

4.6 Fan-triangle padding

Since constructing the padding of several rows of triangles that ignore the ISD is not viable, we instead attempt to make the padding consist solely of border vertices and vertices placed at the intersection. The vertex radii inside the padding had to be kept small to prevent either of the two constraints contesting the shared space close to the intersections. The vertices at the intersection, however, are shared between the constraints, and are as such not bound by the same limitation. Because of this, we may instead attempt to place a set of larger vertex spheres at the intersection, and connect these vertices directly to the border vertices. The spheres at the intersection will essentially function as the padding, and since the bordering vertices follow the ISD mesh density and radius, there should be no strays introduced.

To connect the border vertices with the intersection vertices, we first need to decide the intersection vertex positions and corresponding sphere radii. As mentioned previously, the width of the border itself is not relevant. However, the relation between the width of the border and the density of

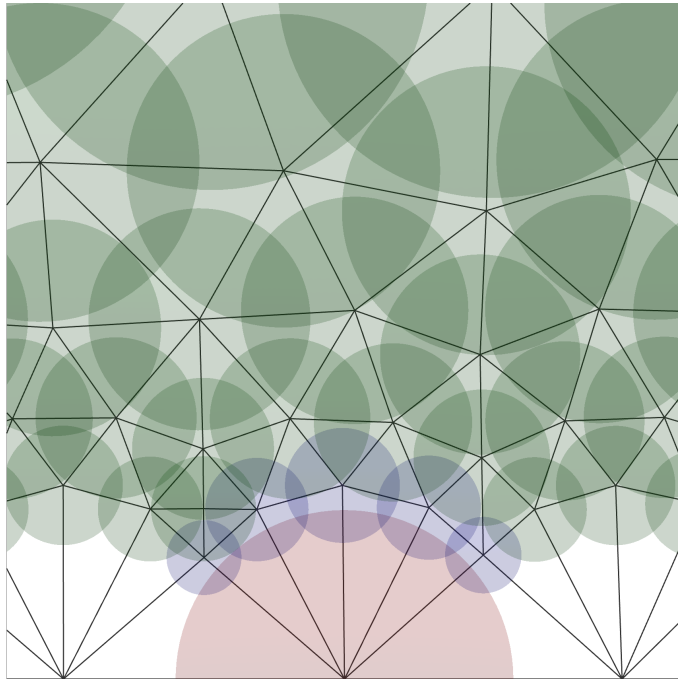


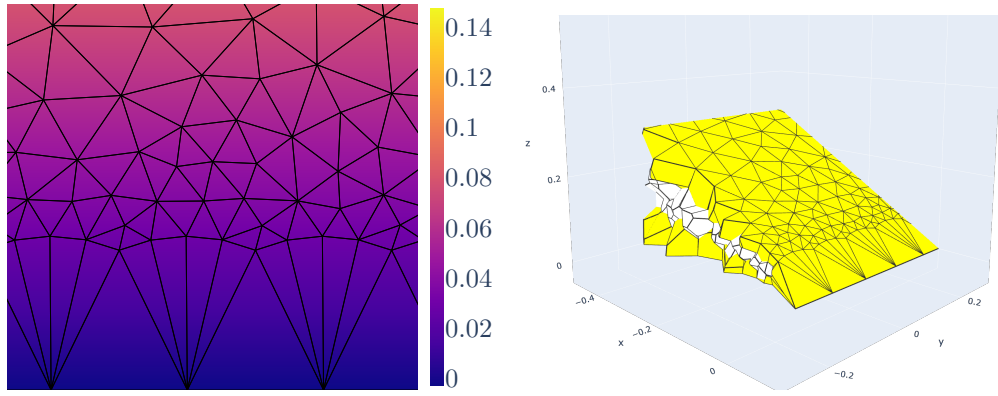
Figure 4.17: Vertex spheres of a main-loop vertex, shown in red. The fan-like construction of the triangles allows the border vertices, shown in blue, to have matching sphere radius to the interior vertices, shown in green. This allows for the sudden jump from an arbitrarily high mesh density to one of significantly lower mesh density.

the intersection vertices is. We set the radii of the intersection vertices to be equal to the distance between them, making the intersections of their spheres form equilateral triangles.

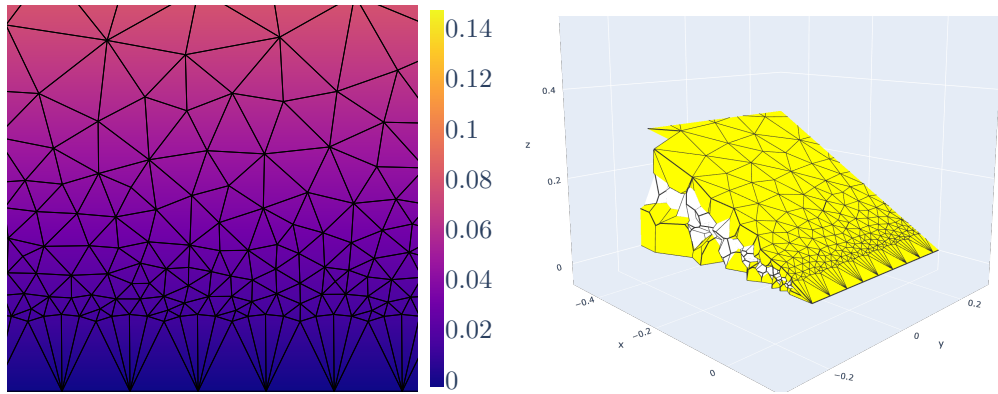
We can place vertices above the tip of each of these triangles, forming part of the padding vertices. We refer to the resulting triangles as main-loop triangles (Figure 4.17). We notice that depending on the distance these vertices have from the sphere intersection, we may increase or decrease the radius of these vertices however we like while still maintaining overlapping with the intersection vertex spheres. As such, we can place these vertices in a position where their radii is set by the ISD.

We notice that these main-loop triangles have spheres too small to intersect one another. To bridge the gap between them, we insert a set of vertices which we will refer to as fan-vertices because of their appearance. These vertices are placed in a curve-like chain spanning the gap between the main-loop vertices. The radius of the fan vertex spheres as well as the distance to their neighbouring fan vertices are decided by the ISD, making their density match that of the interior vertices. The curve is almost circular around the intersection vertex spheres and vary slightly based on the radii of the individual fan-vertices. This results in a set of vertices that intersect the intersection vertices, yet match the mesh density and radii of the vertices in the interior of the plane.

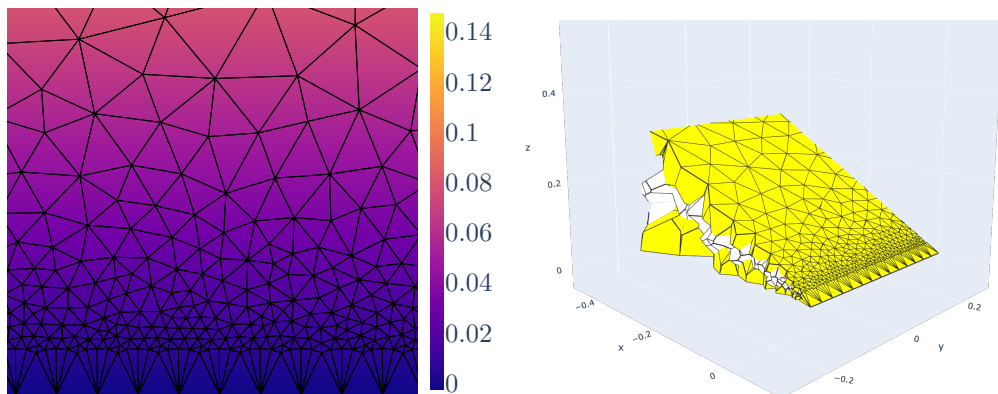
The more aligned the intersecting faces are, the smaller the ISD is at the border, resulting in the fans consisting of more triangles and the triangles being more oblong. This is not ideal, as for sharp intersections, the triangles become less and less uniform with the triangles at the interior. Fixing this, however, is no easy task. As previously mentioned, increasing the density of the main-loop vertices results in a narrower padding with the same ratio of between the width and the fan vertex density. In Figure 4.18, this effect is shown, where the fan vertex padding has been used to bridge the gap between the ISD mesh density and the intersection vertex density. We see that reducing the density results in a narrower boundary with smaller fan triangles arranged in the exact same way.



(a) Density = 0.1

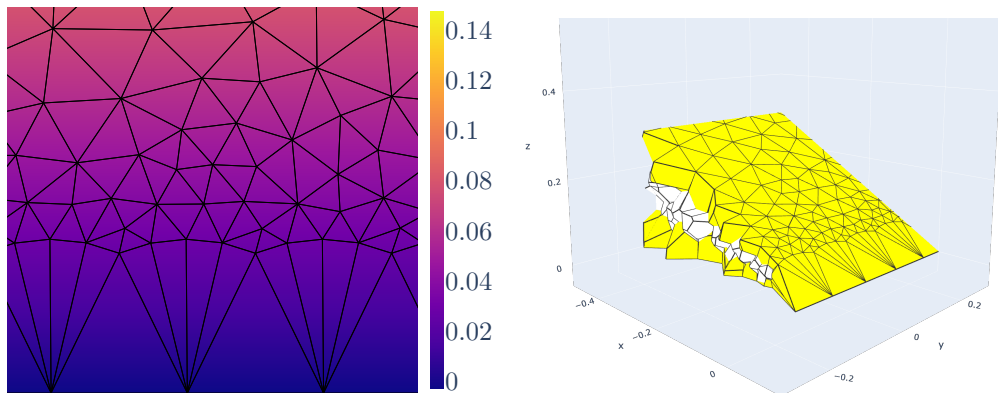


(b) Density = 0.05

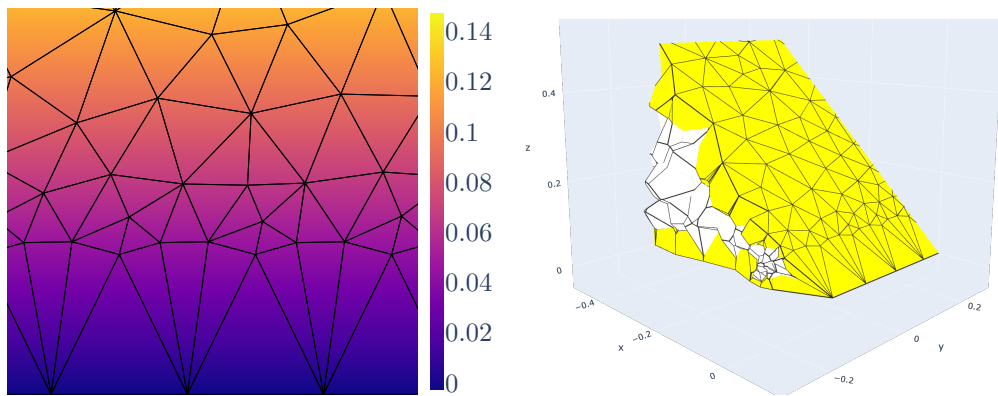


(c) Density = 0.03

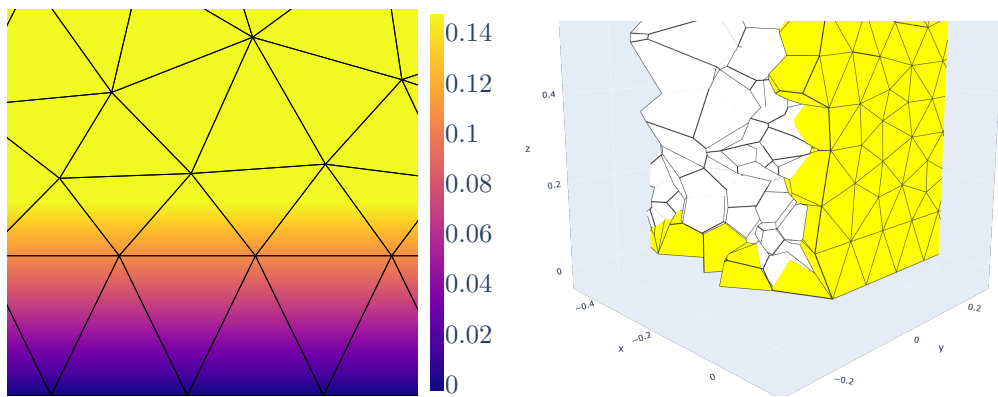
Figure 4.18: Fan triangle padding for a 30° intersection using different densities for the intersection vertices. The ISD across the plane is shown as a heat map(left)



(a) Angle = 30°



(b) Angle = 45°



(c) Angle = 90°

Figure 4.19: Fan triangle padding for intersections at different angles, using the a vertex density of 0.1 at the intersection. The ISD across the plane is shown as a heat map(left)

If the ISD mesh density increases at a higher or lower rate, the shape of the fan triangles will however be altered. In Figure 4.19, the ISD increases at varying rates. This rate is decided by the sharpness of the intersection angle. We see that for sharper angles, the ISD increases more slowly, resulting in smaller mesh sizes at the border.

For the fan triangle padding to work, the designated mesh density at the intersection needs to be

higher than the mesh density at the border. For intersections at 90° or less, this is guaranteed, as the ISD increases by a factor lower than one along the padding width. For more obtuse intersections, this is not the case, and we need a workaround. As mentioned previously, we assign a maximum element size to the mesh to ensure a given level of refinement. By setting the density of the intersection vertices to the same value, the resulting padding will have a width equal to the maximum mesh size. Since the ISD at the obtuse intersection increases by a factor larger than one, the mesh size at the border of the padding will also be clamped to the maximum mesh size, resulting in a padding where the intersection density matches that of the padding density, see Figure 4.19.

It is not necessary to change this default intersection density based on the intersection angle. As shown in Figure 4.18, any density chosen produces the same result, albeit with different triangle sizes. There are, however, other criteria in which the intersection density must be increased.

For the fan triangle padding to work, the intersection vertex spheres need to only overlap their immediate neighbours. If two intersections are close to one another, their spheres must be shrunk to keep them from overlapping. In fact, following the same logic as when designating the plane ISD, we need a distance function that describes the necessary density to avoid overlap with non-neighbouring edges and faces. As such, we need to define the ISD for edges.

4.7 Line ISD

The more complex condition of the main-loop vertices is similar to the condition for the mesh density. For the main-loop vertex spheres to not intersect vertex spheres not belonging to neighbours, their radius also need to be decided by the ISD. This follows the same logic as for the ISD of planes and has its own set of distance functions. When implementing the ISD for a plane, we knew by definition the direction \vec{n}_o from the point \vec{p}_o to the inscribed sphere center \vec{p} had to be the normal of the plane. However, for a given line, the direction \vec{n}_o only has to be orthogonal to the line direction \vec{v} . As a result, three new ISD functions need to be defined for faces, edges and points.

The ISD from a line to a plane is similar to the plane to line ISD defined previously. We first need to find the direction \vec{n}_o from the origin point \vec{p}_o to the inscribed sphere center \vec{p} . For a given \vec{n}_o , the inscribed distance is given by function (4.1). To minimize this distance, we set \vec{n}_o to the orthogonal vector to \vec{v} that minimizes $\vec{n}_o \cdot \vec{n}$, where n is the normal of the plane. We find this vector using the formula $\vec{n}_o^* = (\vec{n} \times \vec{v}) \times \vec{v}$, $\vec{n}_o = \vec{n}_o^* / \|\vec{n}_o^*\|$. Using this value in Equation (4.1) gives us the ISD from a line to a plane.

The calculation of the inscribed distance between two lines poses a significantly more complex challenge. To determine the smallest inscribed sphere between two lines that intersect a given point simultaneously, we aim to minimize Equation (4.3) with respect to any vector \vec{n}_o orthogonal to the direction of the line. In this thesis, we have chosen to omit the direct implementation of this method in favor of an approach that involves sampling a set of orthogonal directions to the line direction and selecting the minimum value. While a direct solution to the problem would be more efficient, considering that this value is only evaluated at the edge vertices, the potential performance gain is negligible.

Finally, the inscribed distance from a line to a point is relatively straightforward. We know from equation (4.5) the inscribed distance to a vertex point for a given normal n_o . We also see that to minimize the distance, $(\vec{p}_o - \vec{p}_v) \cdot \vec{n}_o$ needs to be maximized. Thus, the ISD between an edge and a point is given by the normalized direction \vec{n}_o that maximizes $(\vec{p}_o - \vec{p}_v) \cdot \vec{n}_o$, while being orthogonal to the line direction \vec{d} . We find this by normalizing the vector rejection of $(\vec{p}_o - \vec{p}_v)$ onto \vec{d} ,

$$\vec{n}_o^* = (\vec{p}_o - \vec{p}_v) - ((\vec{p}_o - \vec{p}_v) \cdot \vec{d})\vec{d}, \quad \vec{n}_o = \vec{n}_o^* / \|\vec{n}_o^*\|_2. \quad (4.6)$$

When calculating the ISD for the main-loop vertices, the distance to any plane included in the intersection is ignored, but not its edges. Since the vertex spheres may intersect spheres from other edges of the same plane, these still needs to be taken into consideration. This, however, results in

a familiar problem, where the density of the main-loop vertices approach infinity towards the ends of every edge.

We solve this in a similar fashion to the intersection vertices. We assign a larger vertex radius to the corner, bridging the density gap. The vertices neighbouring the corner are positioned so that their spheres intersect the corner spheres. Like previously, since the density of the main-loop vertices grow inversely with the distance from the corner, the size of these corner spheres does not affect the ratio between the corner sphere radius and the radius of its immediate neighbours. This essentially means we are free to shrink these spheres whenever necessary. Similar to planes and lines, the determination of this size relies on an ISD function.

4.8 Point ISD

The computation of the Inscribed Sphere Distance (ISD) for a point is notably simpler compared to the implementation for triangles and lines. As the set of potential directions N encompasses any normalized vector in \mathbb{R}^3 , the ISD is directly calculated as half the distance between the point and the various element types. Previously, we have discussed the computation of distances between a point and a plane, as well as between a point and a line. Additionally, the distance between two points is determined straightforwardly using the Euclidean distance formula. Similar to the previous implementations, the point of contact determines the applicable distance methods. It is important to note that any neighbouring plane or line to the given point is excluded from the distance calculations.

Chapter 5

Implementation

The implementation of the mesh generation can be found on GitHub[11]. It is separated into three implemented components, covering different functionality necessary to produce our Voronoi mesh. The `ConstrainedEdgeCollection` class is responsible for triangulating a given set of constraint faces and edges, as well as designating some of the vertex radii. The result is a set of triangles covering the surfaces. The triangle density is calculated using the `InscribedSphereField` class. This class calculates the inscribed sphere distance for a given plane and constraints. The triangulation is passed on to the `TriangulatedSurface` class, which is responsible for designating the remaining vertex radii and generating the constrained site location. Finally, the mesh volume is filled with unconstrained sites using the function `add_background_sites`. This function utilizes Gmsh to fill the remaining space with unconstrained sites. To maintain the face constraints any unconstrained stray site is simply removed. The final mesh is constructed using the Scipy Voronoi class.

5.1 ConstrainedEdgeCollection

The `ConstrainedEdgeCollection` (CEC) class is responsible for the triangulation of the constraint faces. It takes as input a set of faces described by their border points and discretizes them into triangles using ISD as a density function as well as adding fan-triangle padding to any shared edge.

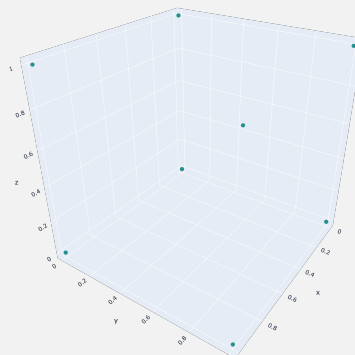
To represent the constraint faces, the CEC needs the position of the points defining its boundary. We refer to these as corner vertices. These are added to the CEC using the function `add_vertices`, which inputs the coordinates of the points and returns the indices of the vertices in the collection.

```
import numpy as np
from pebi_gmsh.constraints_3D.constrained_edges import (
    ConstrainedEdgeCollection
)

CEC = ConstrainedEdgeCollection()

# Defining corner vertices coordinates
# Standard unit cube
cube_points = np.array([
    [0,0,0],
    [1,0,0],
    [1,1,0],
    [0,1,0],
    [0,0,1],
    [1,0,1],
    [1,1,1],
    [0,1,1],
])

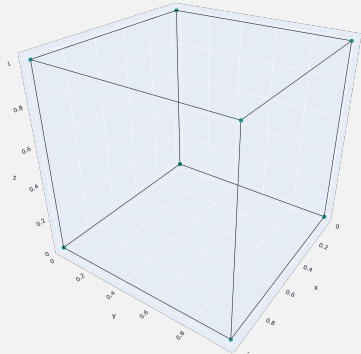
cube_idx = CEC.add_vertices(cube_points)
```



The function `add_face` adds a constraint face to the CEC. It takes the indices of the vertex as input. It is assumed that the vertices of the face are planar and are added in a counter-clockwise order. This is necessary for the face normal to be calculated correctly. The CEC automatically adds the resulting edges to an internal array of edge indices.

```
# cube_idx = [0,1,2,3,4,5,6,7]
cube_idx = CEC.add_vertices(cube_points)

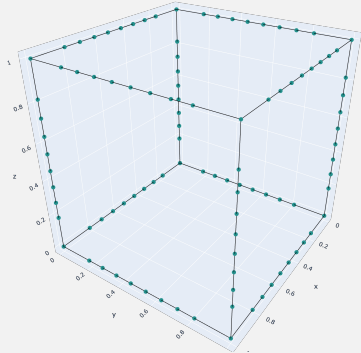
face_vertices = [
    [3,2,1,0], #bottom
    [4,5,6,7], #top
    [0,4,5,1], #Sides
    [1,5,6,2],
    [2,6,7,3],
    [3,7,4,0],
]
for face in face_vertices:
    CEC.add_face(face)
    :
    :
```



It is important that the same vertex id is used by faces that share a vertex. Adding the same coordinate multiple times for the different faces makes it so that the CEC does not recognise which faces share edges and vertices. This in turn prevents it from constructing edge padding correctly, as it assumes the faces are individual, infinitesimally close constraints rather than being connected. It is also worth noting that the CEC is not designed to discover intersections between constraint planes. If two input planes intersect, the intersection has to be represented by an edge in the CEC.

Once the necessary faces have been input, we use the function `populate_edge_vertices` to add intermediate vertices between the corner vertices of every edge. It does this according to the density given by the class `LineInscribedField`. This class is initialized with an origin line and a set of constraint faces, lines and points. It returns the inscribed sphere distance for any point on the origin line using the `distance` function. Each edge is filled with vertices with varying density. Additional space is added at the start and end to account for the radius of the corner vertex spheres.

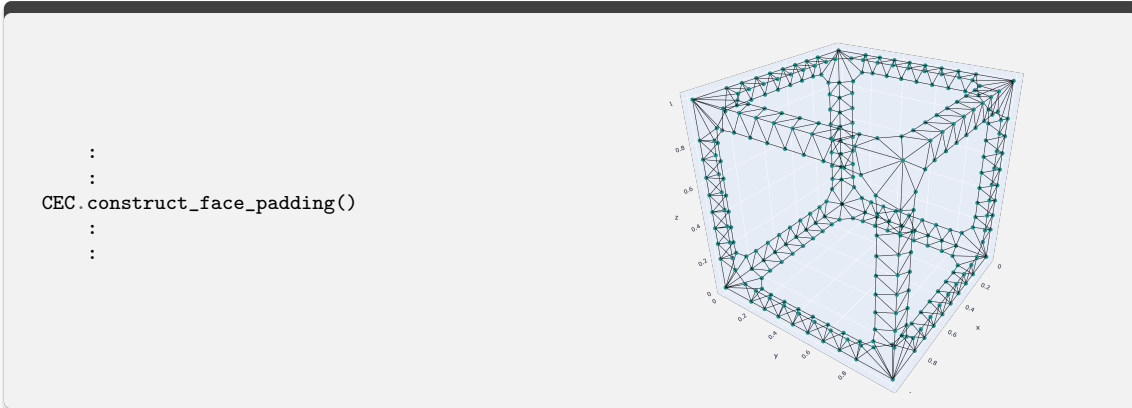
```
:
:
CEC.populate_edge_vertices()
:
:
```



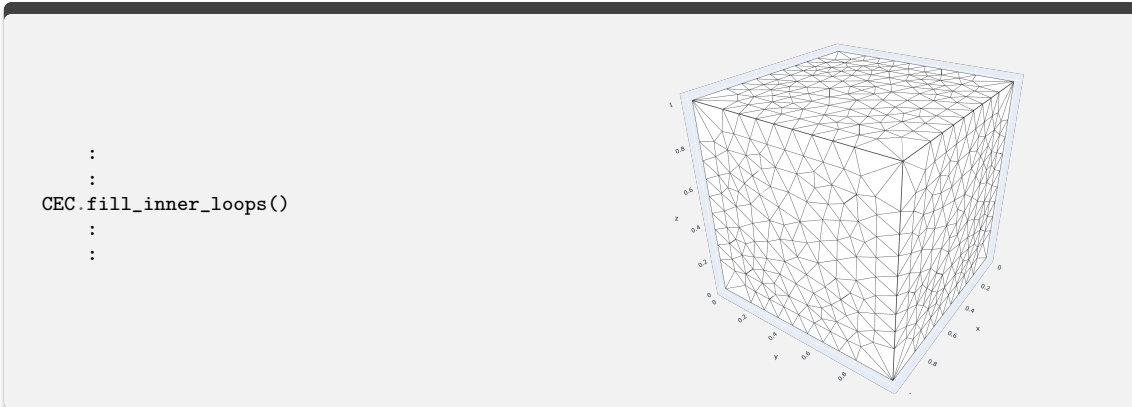
Once the edges have been populated with vertices, the edge paddings need to be constructed. This is done using the function `construct_face_padding`, which either constructs the padding of all the faces or a face given by a face id.

The padding method first constructs the vertices at the intersections of the main-loop vertices. It then fills in each gap with a set of vertices. The number of vertices in each fan as well as their distribution within the fan are decided by the class `InscribedSphereField`, which calculates the inscribed sphere distance of a given point. Just like when using the `LineInscribedField`, the class needs to be instantiated uniquely for each face, as its calculations are based on the position and

normal of the origin plane. The number of fan vertices is calculated using the inscribed sphere distance at points evenly sampled across each fan arch. Once the number of vertices are decided, they are spread across the arc using an iterative spring method comparing their density with that of the ISD at their position.



Finally, the interior of the constraint faces are filled. This is done using Gmsh with a mesh size decided by the ISD. Consult Chapter 6 for more detail about the Gmsh implementation.



This results in a triangulation of the surfaces that serves as the target ridges for the final mesh. This is passed on to the `TriangulatedSurface` class.

5.2 `TriangulatedSurface`

The `TriangulatedSurface` class is responsible for finding a set of radii for the vertex spheres that do not break the face conditions, see Definition 5 and 6. Additionally it is responsible for producing the constrained sites once a valid set of radii is found.

The construction of a valid radii set starts with an initial set of radii that satisfy the first face condition. This is done by assigning each vertex a radius based on its median distance to its neighbours. In theory, the ISD could be used to determine these radii as well, but in practice the discretized triangles will occasionally have too large intermittent variations in size and shape. If a triangle with uneven edge sizes is assigned an even set of vertex radii, there is a significant possibility that one of the spheres gets assigned a radius larger than its max value; see Figure 4.4. As a result, when assigning the vertex sphere radii, it is beneficial to take these variations into consideration.

The padding vertices have initial radii decided by the `ConstrainedEdgeCollection` class in order to ensure they intersect properly with their neighbours. As such, triangles at the border of paddings make the transition from radii decided by the ISD to radii decided by intermittent distance. This

can often be an abrupt transition, leading to initial radii that is either too large or too small. Whenever this occurs, an alternate approach to initial radii is utilized for the given triangle.

A line is drawn normally from the mass center of the triangle. A height is designated as the furthest intersection between the vertex spheres and the line. If no sphere intersects the line, the height is instead set by the median distance to the triangle center.

An intersection point is placed at the given height normally from the mass center. The radii of the triangle vertices are increased to intersect at this point. Since the point by definition lies outside all the vertex sphere, we know this will not shrink any vertex and prevent intersections. We also guarantee that the triangle intersects properly, since the intersection point has been constructed.

Once the initial vertex radii are set, the vertices are each assigned a minimum radii. This is calculated using the function `calculate_minimum_radii`. The values will be used to ensure that we do not retroactively break the first face condition by shrinking the vertex radii too much. Whenever we update any vertex radii, the corresponding minimum radii are updated as well.

To enforce the second face conditions, an algorithm iterates through each vertex. If any sites are inside the vertex spheres the radius is shrunk. In order to prevent the stray, the vertex radius needs to be shrunk past a necessary threshold. We could shrink the sphere precisely down to this value, but this would result in odd cells that barely maintain the face condition. Instead, the vertex radii are shrunk to the halfway point between the threshold and the minimum radii.

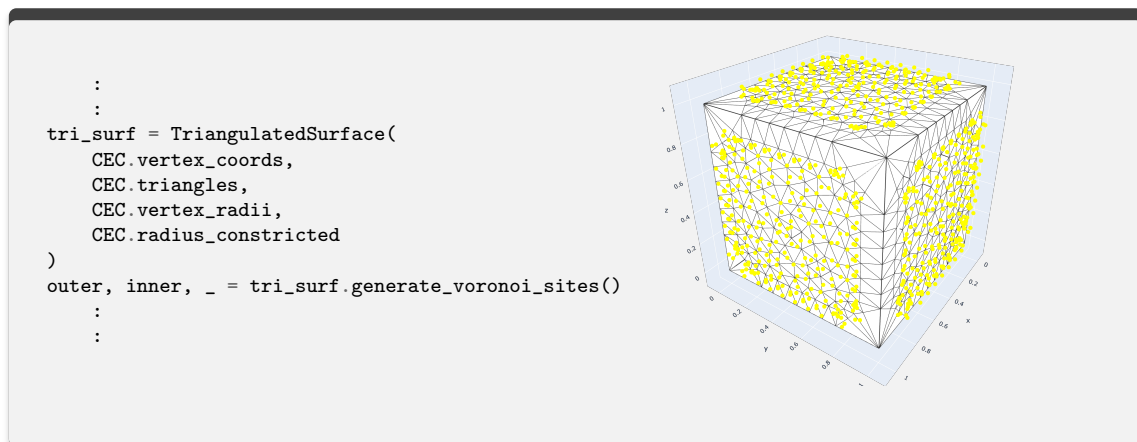
It is common for the threshold to be smaller than the minimum radius of a given vertex. In these cases there is nothing that can be done by only shrinking the vertex itself. The `TriangulatedSurface` class uses two methods to resolve this.

The first method decides the order in which the vertices are iterated. The vertices are sorted by the difference between their current and minimum radii. This way, vertices with more leeway are shrunk first. The stray sites typically occur in groups, caused by some sphere that is too large for its surrounding area. This method essentially shrinks such outliers first, instead of forcing the surrounding vertices to resolve the issue individually.

The second method is simply to run the iteration more than once. As previously mentioned, the stray sites can often be resolved by other vertices being shrunk. Additionally, some strays get moved away from a vertex as a result of some other strays resolution, this can simplify or even resolve the stray, meaning a shrinking threshold may be more achievable in later iterations.

Because of the construction of the vertex position, these methods are typically sufficient to resolve all stray sites. However, if the algorithm encounters any unsolvable strays in its final iteration, an error message is sent.

Once the algorithm is finished, the site locations are generated using the function `generate_voronoi_sites`.



5.3 Background sites

With the construction of the constrained sites complete, we could create a Voronoi mesh that satisfies all our constraint criteria. However, the interior would then be filled with elongated cells extruding from the constraints.

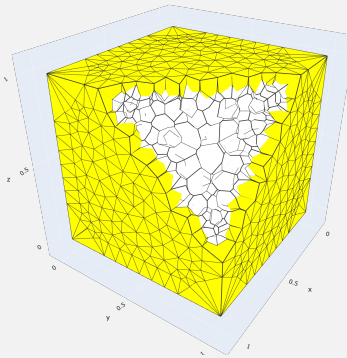
We use the function `add_background_sites` to fill the interior space. This utilizes Gmsh to distribute the sites inside the given mesh. In order to smooth out the transition between the constrained sites and the newly generated ones, we embed the constrained sites into the mesh, meaning Gmsh is essentially forced to construct a set of nodes that includes the old site locations.

The density of background sites is determined by a mesh size field; see Section 6.1. The field decreases the density of sites in areas further away from the constrained sites. This results in larger cells further away from the constraints.

Additional stray sites can be introduced by the generation of background sites. However, since these sites are not required to maintain the constraints, we simply remove them.

Finally, we generate the Voronoi mesh using the Scipy [20] class `Voronoi`. This class utilizes the QHull algorithm [3] to generate the final mesh from the site locations.

```
sites = np.vstack((outer, inner))
background_sites = add_background_sites(
    CEC.vertex_coords,
    CEC.edge_corners,
    np.vstack(CEC.constraint_tris),
    CEC.face_edges,
    sites,
    tri_surf.vertex_radii
)
voronoi = Voronoi(
    np.vstack((sites, background_sites))
)
:
:
```



Chapter 6

Gmsh Implementation

This chapter will go through some of the basics of Gmsh, as well as some of its more advanced features that have been utilized in this project. For a more detailed overview of Gmsh and its API, we refer you to the Gmsh reference page [9], which contains API descriptions and tutorials.

6.1 Basics

Gmsh is a finite element mesh generator that specializes in triangular and quadrilateral meshes in 2D and 3D. The engine itself can be used through its own graphical user interface or through API for a collection of different programming languages. We will be utilizing the Python API.

```
import gmsh
import numpy as np

# Initializing gmsh
gmsh.initialize()
gmsh.model.add("Demo")
```

After initializing Gmsh, we need to assign some boundary for it to fill. We start by defining the points of the boundary using the function `gmsh.model.geo.add_point`.

```
# Point coordinates
# Must be in 3D, even if all z values are 0
boundary_points = np.array([
    [0,0,0],
    [1,0,0],
    [1,1,0],
    [0,1,0],
])

# Ids/tags of the points added
boundary_point_tags = []

#Adding the points
for point in boundary_points:
    tag = gmsh.model.geo.add_point( x = point[0], y = point[1], z = point[2])
    boundary_point_tags.append(tag)
```

We then construct edges using the IDs, also known as tags, of the points. All the geometry we add has tags, and we may either assign them manually or get them as return values offunctions we use to add geometry. We then construct a curve loop from these edges. Gmsh has several types of curves that we can use to construct curves, such as circle segments or splines. It places

no restriction on the combination of curves used to create the loop, though it is important that each curve starts at the end point of the previous one. This even accounts for direction. Assigning negative curve tags to the loops signifies that the loop traverses the curve backwards.

```
# End points of the line segments
edge_point_tags = np.array([
    [1,2],
    [2,3],
    [3,4],
    [4,1],
])

# Tags of the border edges
border_edge_tags = []
for edge in edge_point_tags:
    tag = gmsh.model.geo.add_line(edge[0], edge[1])
    border_edge_tags.append(tag)
```

Last, we define a plane surface by a list of loop tags. The first loop defines the main boundary, and the rest define holes.

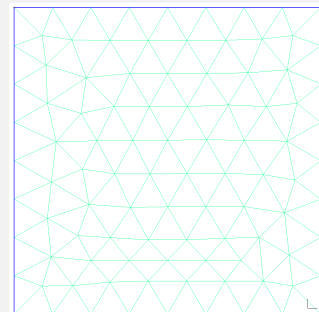
```
# Adding the surface plane
border_loop_tag = gmsh.model.geo.add_curve_loop(border_edge_tags)
surface_tag = gmsh.model.geo.add_plane_surface([border_loop_tag])
```

To utilize the added geometry, we run the `geo.synchronize` function. This essentially updates Gmsh with all the information we have added. We then run the function `mesh.generate`, which takes the number of dimensions as a parameter, and get the surface discretized into triangles.

```
# Updated the added geometry
gmsh.model.geo.synchronize()

# Generates the 2D mesh
gmsh.model.mesh.generate(2)

# Visualizes the mesh in the Gmsh GUI
gmsh.fltk.run()
```



We wish to export the generated nodes and triangles back to Python. To retrieve the resulting points and triangles we need to run the function `mesh.create_faces` to add the generated faces to Gmsh's internal indexing. This allows us to get the coordinates of the nodes and the tags of their corresponding triangles.

```
import matplotlib.pyplot as plt

gmsh.model.mesh.create_faces()

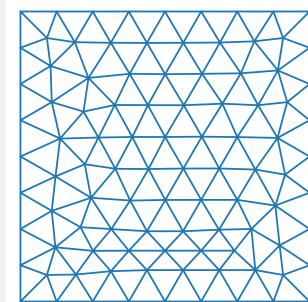
node_tags, node_coords, node_params = gmsh.model.mesh.get_nodes()
tri_tags, tri_node_tags = gmsh.model.mesh.get_all_faces(3)

# The tags and coordinates are returned as a flattened array
node_coords = node_coords.reshape(-1,3)
```

Here, it is worth noting that Gmsh indexing starts at one, so we need to subtract one from the indices to work with packages like Numpy.

```
# The indices must be subtracted by one to start at 0
tri_node_tags = tri_node_tags.reshape(-1,3) - 1

plt.triplot(
    node_coords[:,0],
    node_coords[:,1],
    triangles = tri_node_tags
)
plt.show()
```

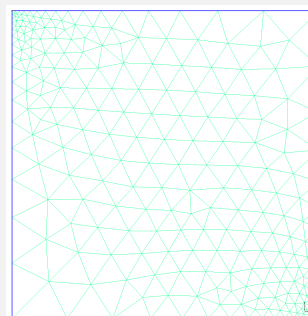


To generate 3D meshes, we essentially do the exact same approach, except with an additional step. Just like we constructed a surface from a curve loop, we need to construct a volume from a surface loop. The surface loop needs to be a collection of surfaces that cover the volume in question, like the sides of a cube. There exists many types of surfaces one can construct in Gmsh, but for our purposes, planar surfaces constructed from line segments are sufficient.

6.2 Mesh size

So far, we never specified the mesh resolution and it was thus determined by the default value (0.1) passed to new points. Whenever Gmsh tessellates a surface or volume, it uses a mesh size that describes the target value of its edges to determine the level of refinement. We can customize how Gmsh calculates this mesh size to generate meshes with varying refinement. The simplest way is to alter the mesh sizes given to the corner points of our mesh. By default, Gmsh interpolates across these values to determine the mesh size.

```
mesh_size = [0.2, 0.01, 0.2, 0.01]
for point, size in zip(boundary_points, mesh_size):
    tag = gmsh.model.geo.add_point(
        point[0], point[1], point[2], meshSize = size
    )
    boundary_point_tags.append(tag)
```



If we require more nuanced methods of determining mesh sizes, we can utilize the mesh size fields in Gmsh. These are mathematical fields which evaluate the mesh size at any given point in the space. Gmsh has a multitude of pre-defined fields implemented. These fields can also be combined into other fields with the field functions such as Add, Subtract, Min, Max and MathEval. These combination methods are important, as the mesh can only have one mesh size field, which is set by the function `mesh.field.set_as_background_mesh`. If we want a combination of factors to affect the mesh size, we can combine several mesh-fields into one. For the mesh field to solely define the mesh size, some default options of Gmsh need to be turned off.

```

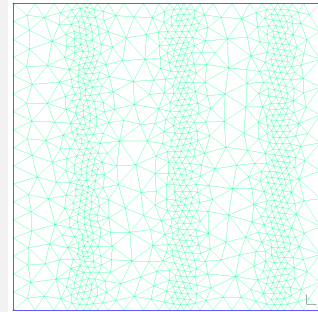
# Initialing gmsh
gmsh.initialize()
gmsh.model.add("Demo")

# Turns off default mesh size calculations
gmsh.option.setNumber(
    "Mesh.MeshSizeExtendFromBoundary", 0
)
gmsh.option.setNumber(
    "Mesh.MeshSizeFromPoints", 0
)
gmsh.option.setNumber(
    "Mesh.MeshSizeFromCurvature", 0
)

# Adding a MathEval field and assign its function
math_field_tag = gmsh.model.mesh.field.add("MathEval")
gmsh.model.mesh.field.set_string(math_field_tag, "F",
    "0.04 * Sin(x * 20) + 0.06")

# Setting the field as the background field
gmsh.model.mesh.field.setAsBackgroundMesh(math_field_tag)

```



6.3 External process mesh-field

While the predefined mesh size field in Gmsh is typically more than sufficient for most purposes, we require a bit more flexibility to implement ISD-based mesh sizes. While the MathEval field allows us to specify mathematical expressions, the entire expression has to be represented by a string using the built-in syntax defined by Gmsh, and it is too limiting for our purposes.

Luckily, Gmsh has a mesh size field that passes the computation of mesh size to an external process, aptly named the ExternalProcess field. This field passes single instances of 3D coordinates to an external program, in our case Python, and waits for the resulting mesh size to be returned. The individual coordinates are written to a system buffer and so are the returned mesh sizes. This allows us to use the InscribedSphereField class to calculate the mesh size of planes.

Example of a .py file that could be used as an ExternalProcess field.

```

import struct
import sys
import math

# Here we may initialize the ISD
# field = InscribedSphereField(normal, tris)

# Loops until Gmsh feeds the process a nan
while(True):
    # Reads the coordinates fed by Gmsh
    xyz = struct.unpack("ddd", sys.stdin.buffer.read(24))
    if math.isnan(xyz[0]):
        break
    # Here any logic can be added
    # f = field.distance(xyz)

    f = 0.001 + xyz[1]*0.009
    sys.stdout.buffer.write(struct.pack("d",f))
    sys.stdout.flush()

```

Example of initializing an ExternalProcess field.

```
external_field_tag = gmsh.model.mesh.field.add("ExternalProcess")
gmsh.model.mesh.field.set_string(external_field_tag,
    "CommandLine",
    "python " + "path/to/the/file.py"
)
gmsh.model.mesh.field.setAsBackgroundMesh(external_field_tag)
```

There are, however, a few limitations to the performance of this approach. The mesh size is evaluated thousands of times for each surface, increasing with smaller mesh sizes as more points need to be distributed. As a result, the function for evaluating the mesh size needs to be fast. Since the input is given one point at a time, vectorizing the expression for arrays of points is not possible. Using Numba to compile the ISD functions seemed to sufficiently speed up the function for our purposes.

Additionally, since the InscribedSphereField is unique to each plane discretized, we need to discretize each plane individually instead of adding them all to Gmsh and discretizing them in one large batch.

6.3.1 Meshing algorithms

Gmsh has several different meshing algorithms to choose from. The default **Frontal-Delaunay** results in the highest cell quality, at a slight cost in performance. However, the **Delaunay** algorithm works better when solely relying on the mesh size fields. For our purposes this makes the **Delaunay** algorithm preferable.

```
gmsh.initialize()
gmsh.model.add("Demo")

# Sets the meshing algorithm
# 1: MeshAdapt, 2: Automatic, 3: Initial mesh only, 5: Delaunay
# 6: Frontal-Delaunay, 7: BAMG, 8: Frontal-Delaunay for Quads
# 9: Packing of Parallelograms, 11: Quasi-structured Quad
gmsh.option.setNumber("Mesh.Algorithm", 5)
```

6.4 Embedding

When adding the background sites, we would like them to be positioned well alongside the constrained sites. We have already decided that these sites will be included in the mesh. The remaining sites should therefore be placed assuming these sites are already placed.

This can be achieved with the `mesh.embed` function. This function adds lower-level geometry, such as lines and points and faces, to volumes and surfaces. This works similarly to how borders are defined. For instance, the embedded lines and surfaces can be subdivided according to the mesh size. The local mesh size can also be calculated by the embedded features, just like they were for the boundary using the default Gmsh settings.


```

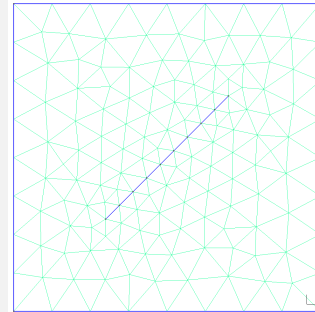
line_pos = np.linspace(0.3,0.7, 10, endpoint=True)\
    .reshape(-1,1) * np.array([1,1,0])

line_point_tags = []
for point in line_pos:
    tag = gmsh.model.geo.add_point(
        point[0], point[1], point[2]
    )
    line_point_tags.append(tag)

line_tags = []
for i in range(len(line_point_tags)-1):
    tag = gmsh.model.geo.add_line(
        line_point_tags[i], line_point_tags[i+1]
    )
    line_tags.append(tag)
# The geometry needs to be updated before embedding
gmsh.model.geo.synchronize()

gmsh.model.mesh.embed(1, line_tags, 2, 1)

```



6.5 Transfinite curves

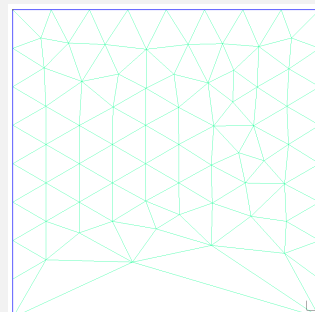
Gmsh will subdivide any curve defining its boundary in order to better match the mesh size. Throughout our implementation, there are cases where we need the boundary edges to be undivided. An example of this is when using Gmsh to fill in the interior of the planes. The boundary in this case is defined by the fan vertices. The fan vertices have been meticulously placed with regard to the local ISD and distance from the main-loop vertices. As such, introducing additional vertices between them would break our implementation.

Gmsh has the option of setting so-called transfinite curves, where we specify the number of points the curve should contain. By setting this value to 2 for the start and the end position, we prevent any subdivision. It is important to note that this can cause terribly misshapen meshes if the transfinite curve density does not match the local mesh density specified by the mesh size field. This is not a problem for the fan vertices, as they are spaced according to the ISD to begin with.

```

# Tags of the border edges
border_edge_tags = []
for edge in edge_point_tags:
    tag = gmsh.model.geo.add_line(
        edge[0], edge[1]
    )
    border_edge_tags.append(tag)
# Sets the bottom edge to only contain 2 points
gmsh.model.geo.mesh.set_transfinite_curve(
    border_edge_tags[0], 2
)

```



Chapter 7

Results and Further Development

One of the primary challenges in constructing constrained Voronoi meshes is dealing with conflicted space. Unlike traditional unstructured meshing methods, where features can be directly embedded into the geometry, constraints in Voronoi meshes are indirectly enforced through the proximity of constrained sites. Consequently, there is a risk of interference between these constrained sites.

To address this issue, we have introduced the inscribed sphere distance (ISD) as a metric for determining the mesh density of the constraint discretization. This innovative approach enables our algorithm to generate constrained Voronoi meshes even in narrow spaces, where conflicts and overlapping of constrained sites would typically occur. By incorporating the ISD metric, we can effectively control the mesh density and achieve accurate and reliable results in challenging geometric scenarios.

An illustrative demonstration of the algorithm's effectiveness is presented in Figure 7.1 and 7.2. Despite the presence of sharp angles and narrow spaces, the algorithm successfully constructs a valid Voronoi mesh. Notably, as the mesh becomes narrower, the resolution of the constraint tessellation increases. This adaptive refinement ensures the generation of well-shaped cells and mitigates the introduction of stray sites.

The implementation of Fan-edge padding at the intersections and corners of the constrained mesh enables the algorithm to successfully generate meshes with sharp angles. The cells pertaining to constrained sites are shown in yellow while the ones pertaining to background sites are shown in white.

7.1 Misshapen fan edges

The implementation of fan-triangle padding effectively addresses the problem of contested space along the edges of the Voronoi mesh. Nonetheless, this approach tends to deform the resulting triangles when dealing with highly sharp edges. Specifically, the triangles become narrower and exhibit a more pronounced distinction between the primary triangles and the fan triangles.

The presence of sharp angles at intersecting features significantly impacts the mesh quality, which is particularly crucial for various applications involving discrete meshes. One notable example is the simulation of porous flow in geological reservoirs[12, 15, 13]. In such simulations, the mesh must accurately represent the intersection points between fractures and sedimentary layers, which are abundant and often occur at sharp angles. Since the simulation heavily relies on the interactions taking place at these feature intersections, maintaining a high mesh quality becomes imperative. Unfortunately, our current algorithm yields cells of subpar quality precisely at these critical intersections, posing a challenge.

The issue at hand cannot be solely attributed to our implementation. As shown in Section 4.5, the underlying cause of this problem stems from the inverse relationship between the required

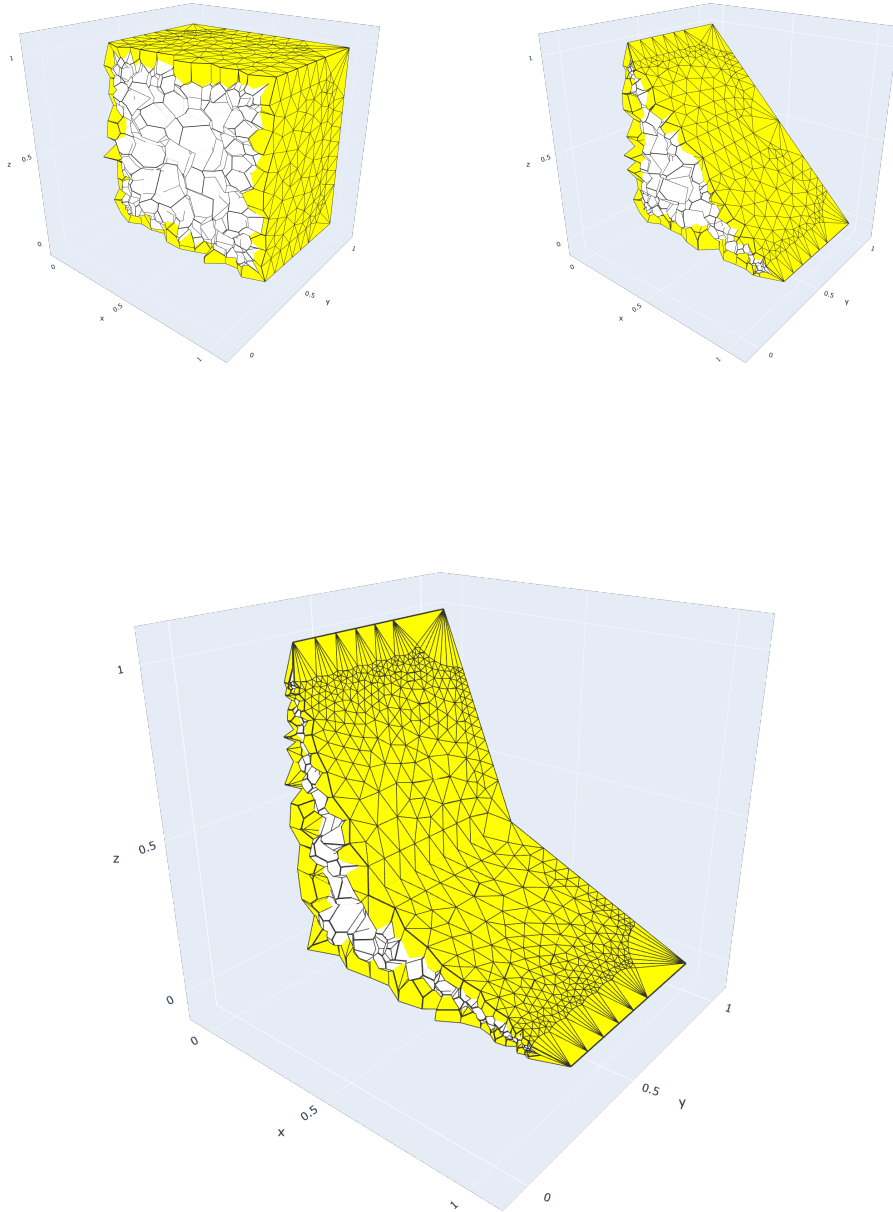


Figure 7.1: Voronoi mesh of a unit cube with one offset edge, causing a gradual sharpening of angles

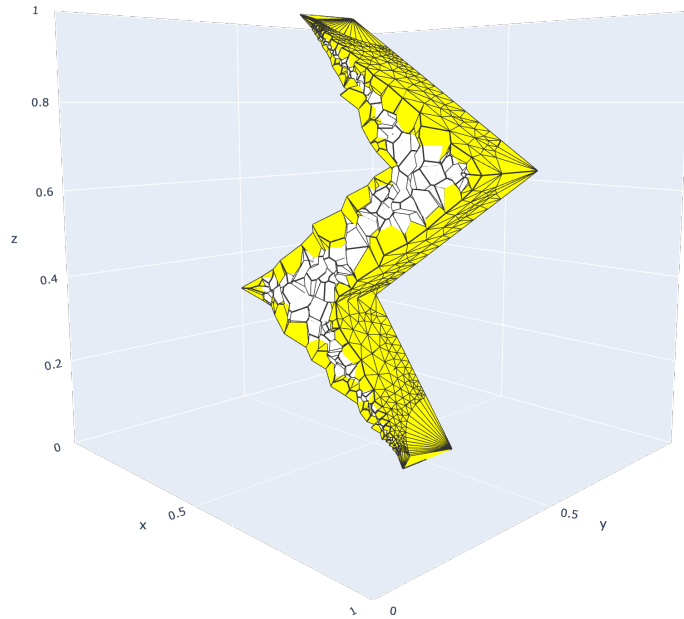


Figure 7.2: Lightning bolt

Chazelle

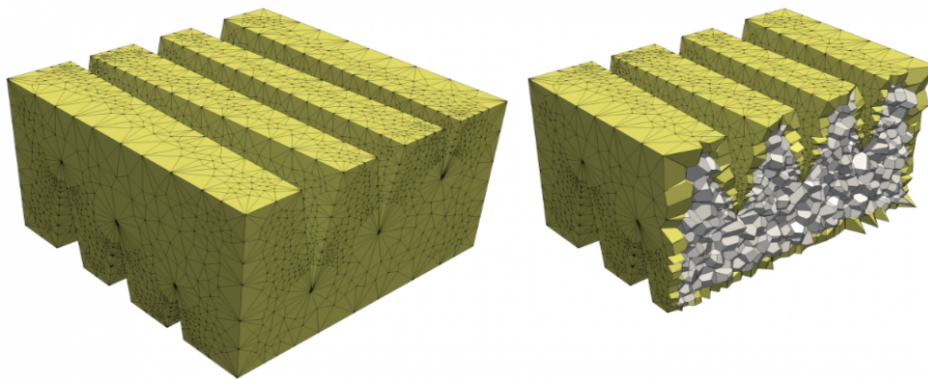


Figure 7.3: Mesh containing sharp edges generated by the Vorocrust algorithm [1]. Image copied from <https://vorocrust.sandia.gov>, 09/06/2023.

mesh density for well-shaped cells and the distance from the intersection. Notably, the width of a properly formed padding is directly proportional to the mesh size at the intersection.

Upon comparing our results with those obtained from the Vorocrust algorithm [1], an interesting trend emerges. We observe that meshes featuring sharp angles exhibit a similar pattern of elongated triangles encompassing these angles; see Figure 7.3. This finding is significant considering that Vorocrust employs a completely different technique for the tessellation of its constraints. In Vorocrust, vertices are recursively added until a valid arrangement of vertex spheres covers the constrained surface. The fact that this methodology fails to generate well-shaped edge triangles suggests that achieving such triangles is either inherently impossible or requires careful construction.

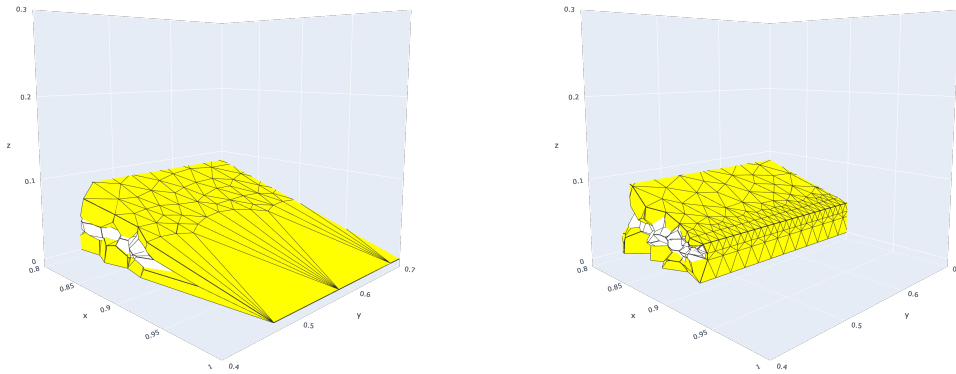


Figure 7.4: The act of clipping a sharp intersection on a constraint surface yields fan-triangles that are better suited to complement the surrounding triangles.

Rather than attempting to resolve this issue solely using Voronoi meshing, an alternative approach would be to take a step back and consider a partial Voronoi solution. The advantages of Voronoi meshing, compared to other polygonal or polyhedral meshes, lie in its ease of construction, especially when dealing with unconstrained cells. This is particularly evident when constructing a mesh that conforms to multiple constraints simultaneously, as the interior of polygonal meshes offers smoother transitions between these various constraints, requiring only an even distribution of site locations.

However, in the vicinity of intersections, our concern is primarily focused on the subset of the intersecting features. In this context, simpler cells like tetrahedral cells becomes a significantly more viable option. Therefore, it may be more feasible to consider the creation of a non-Voronoi padding that specifically handles the transition from the feature intersection to its interior. By adopting this approach, we can potentially achieve a more effective solution.

As depicted in Figure 7.4, it can be observed that the fan-triangles located at the edges of a constraint wedge exhibit a more well-formed shape when the tip of the edge is clipped. By performing this clipping procedure, the intersecting angle is reduced to below 90 degrees, leading to a constraint that our implementation can handle more effectively, as illustrated in Figure 4.19.

The triangles formed along the clipped edge are structurally sound and can be seamlessly connected to the final discretization of the wedge.

7.1.1 Curved constraints

At present, the existing implementation only supports meshes that conform to planar constraints. Consequently, any intersections or connecting edges found within the constraints are treated as sharp edges, resulting in the inclusion of these edges in the mesh geometry. While it is expected for the final mesh to contain edges, an ideal implementation would provide a means to represent a curved surface from which the target vertices can be sampled.

In other words, the current implementation does not yet have the functionality to capture and represent curved surfaces in the resulting mesh. Incorporating this capability would allow for more accurate and realistic mesh generation, as it would enable the sampling of vertices from a continuous curved surface instead of relying solely on sharp edges.

Several approaches can be explored to achieve this goal. One option is to leverage the features of Gmsh, which supports various types of curved edges and surfaces, such as splines or spherical segments. However, integrating these features would require a re-implementation of the ISD mesh density field to accommodate curved surfaces. This task is not straightforward, as the ISD relies

on the normal of the origin plane. Mesh fields utilized by Gmsh, on the other hand, are solely dependent on the position of a given node and do not inherently provide surface normals. Hence, a workaround is necessary to incorporate the required normal information into the process.

Moreover, it is worth noting that while the Gmsh API is user-friendly, it is not universally applicable. Ideally, the input method should be more generalized and adaptable for surfaces imported from other software. One common approach to achieving this is by representing the curved surface using a high-fidelity discrete surface. By using a sufficiently high resolution, the discrete surface can effectively mimic a curved surface and be utilized for sampling the target mesh vertices.

However, it is important to consider that while Gmsh provides functionality for refining geometry, it lacks the ability to coarsen it. Consequently, any high-fidelity surface imported into Gmsh would result in a mesh of equally high resolution. The resulting discretization would include all the geometry of the high fidelity mesh, resulting in a needlessly high resolution Voronoi mesh.

7.1.2 Performance

Currently, the workaround for the lack of normal information involves individually tessellating each constraint surface. However, this approach incurs a significant performance cost since initializing Gmsh for each surface is required. Consequently, the current implementation is not a viable option for handling large numbers of constraints.

As mentioned earlier, if a workaround can be devised to incorporate surface normals as input from Gmsh, the tessellation of constraint surfaces could be batched together. This would enhance efficiency and enable the handling of a larger number of constraints.

Moreover, modifying the mesh density field to utilize the surface normal as an input is essential. The current implementation relies on the normal to cache a set of helper variables during initialization. However, with the inclusion of surface normals, these variables would need to be calculated at runtime instead.

In summary, finding a solution to incorporate surface normals from Gmsh as input, along with necessary modifications to the mesh density field, would streamline the process by allowing batched tessellation of constraint surfaces and eliminating the performance overhead caused by initializing Gmsh for each surface.

7.2 Improvements to ISD

While the ISD has shown itself capable of deciding the necessary mesh density for constrained Voronoi generation, there are examples where the notions behind it fail. While these cases have not shown up in testing so far, they are still possible.

In Figure 7.5, an example is presented to demonstrate a particular challenge. The placement of a disconnected constraint results in a disproportionately large Inscribed Sphere Distance (ISD) at a relatively short distance. Regardless of the positioning of the sites surrounding the small constraint segment, the ISD function causes the segment to fall within the interior of the larger vertex sphere. This highlights the need to address the issue of excessively large ISD values in close proximity.

By incorporating Lipschitz continuity into the ISD field, it is possible to mitigate the increase in radius that occurs within the mesh. This continuity constraint ensures that the ISD values between neighboring points do not differ by a greater proportion than their corresponding distances. In other words, the change in ISD is limited in relation to the distance between points, resulting in a more controlled and gradual transition in the radius values across the mesh. This approach helps maintain the stability and integrity of the mesh generation process while avoiding abrupt variations in the radius values that could adversely affect the accuracy of the simulation.

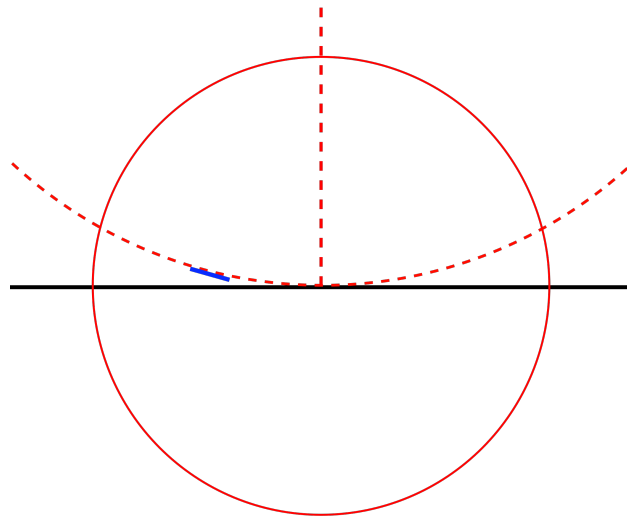


Figure 7.5: A small constraint segment(blue) placed inside the vertex sphere(solid red). The ISD(dotted red) at a nearby location is large as a result of its unusual placement and small size

Bibliography

- [1] Ahmed Abdelkader et al. ‘VoroCrust: Voronoi Meshing Without Clipping’. In: *ACM Trans. Graph.* 39.3 (May 2020). ISSN: 0730-0301. DOI: 10.1145/3337680. URL: <https://doi.org/10.1145/3337680>.
- [2] Nina Amenta and Marshall Bern. ‘Surface Reconstruction by Voronoi Filtering’. In: *Discrete & Computational Geometry* 22 (Dec. 1999), pp. 481–504. DOI: 10.1007/PL00009475.
- [3] C. Bradford Barber, David P. Dobkin and Hannu Huhdanpaa. ‘The Quickhull Algorithm for Convex Hulls’. In: *ACM Trans. Math. Softw.* 22.4 (Dec. 1996), pp. 469–483. ISSN: 0098-3500. DOI: 10.1145/235815.235821. URL: <https://doi.org/10.1145/235815.235821>.
- [4] Runar L. Berge. *Unstructured PEBI-grids Adapting to Geological Features in Subsurface Reservoirs*. NTNU, June 2016. URL: <https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2411565>.
- [5] Runar L. Berge, Øystein S. Klemetsdal and Knut-Andreas Lie. ‘Unstructured PEBI Grids Conforming to Lower-Dimensional Objects’. In: *Advanced Modeling with the MATLAB Reservoir Simulation Toolbox*. Cambridge University Press, Nov. 2021, pp. 3–45. DOI: 10.1017/9781009019781.005. URL: <https://doi.org/10.1017/9781009019781.005>.
- [6] *Computational Geometry: Algorithms and Applications*. Springer, 2008, pp. 147–171.
- [7] Qiang Du, Vance Faber and Max Gunzburger. ‘Centroidal Voronoi Tessellations: Applications and Algorithms’. In: *SIAM Review* 41.4 (1999), pp. 637–676. DOI: 10.1137/S0036144599352836. URL: <https://doi.org/10.1137/S0036144599352836>.
- [8] C. Geuzaine and J.-F. Remacle. ‘Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities.’ In: *International Journal for Numerical Methods in Engineering* 79(11) (2009), pp. 1309–1331.
- [9] *GMSH Reference page*. <https://gmsh.info/doc/texinfo/gmsh.html>. Accessed: 2023-06-04.
- [10] Sondre Husøy. *Constrained Generation of Voronoi Meshes using GMSH*. NTNU, Dec. 2022.
- [11] Sondre Husøy. *Constrained Voronoi generation using ISD*. https://github.com/SondreHus/upr_gmsh. Version 1.0.0. June 2023.
- [12] *Unstructured Gridding and Consistent Discretizations for Reservoirs with Faults and Complex Wells*. Vol. Day 3 Wed, February 22, 2017. SPE Reservoir Simulation Conference. D031S009R005. Feb. 2017. DOI: 10.2118/182666-MS. eprint: <https://onepetro.org/spersc/proceedings-pdf/17RSC/3-17RSC/D031S009R005/1290086/spe-182666-ms.pdf>. URL: <https://doi.org/10.2118/182666-MS>.
- [13] P. Lemonnier and Bernard Bourbiaux. ‘Simulation of Naturally Fractured Reservoirs. State of the Art’. In: <http://dx.doi.org/10.2516/ogst/2009067> 65 (Mar. 2010). DOI: 10.2516/ogst/2009066.
- [14] Bruno Lévy and Yang Liu. ‘Lp Centroidal Voronoi Tessellation and Its Applications’. In: *ACM Trans. Graph.* 29.4 (July 2010). ISSN: 0730-0301. DOI: 10.1145/1778765.1778856. URL: <https://doi.org/10.1145/1778765.1778856>.
- [15] Shahid Manzoor, Michael Edwards and Ali Dogru. ‘Three-dimensional unstructured gridding for complex wells and geological features in subsurface reservoirs, with CVD-MPFA discretization performance’. In: *Computer Methods in Applied Mechanics and Engineering* 373 (Jan. 2021), p. 113389. DOI: 10.1016/j.cma.2020.113389.

-
- [16] Fadl Moukalled, Luca Mangani and Marwan Darwish. *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM® and Matlab®*. Vol. 113. Oct. 2015. ISBN: 978-3-319-16873-9. DOI: 10.1007/978-3-319-16874-6.
- [17] Stefano Rebay. ‘Efficient unstructured mesh generation by means of Delaunay triangulation and Bowyer-Watson algorithm’. In: *1993, Journal of Computational Physics* (1993), pp. 25–138.
- [18] Knut-Andreas Lie Runar L. Berge Øystein S. Klemetsdal. ‘Unstructured PEBI Grids Conforming to Lower-Dimensional Objects’. In: *Advanced Modeling with the MATLAB Reservoir Simulation Toolbox* (2021), pp. 3–45.
- [19] L. Ridgway Scott Susanne C. Brenner. *The Mathematical Theory of Finite Element Methods*. New York, NY: Springer, 2007. DOI: <https://doi.org/10.1007/978-0-387-75934-0>.
- [20] Pauli Virtanen et al. ‘SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python’. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.
- [21] Emo Welzl, Peter Su and Robert Drysdale. ‘A Comparison of Sequential Delaunay Triangulation Algorithms.’ In: *Comput. Geom.* 7 (Jan. 1997), pp. 361–385.



 **NTNU**

Norwegian University of
Science and Technology